



LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FACULTY FOR MATHEMATICS, COMPUTER SCIENCE AND STATISTICS

MASTER THESIS

Extending Hyperband with Model-Based Sampling Strategies

Author:

Niklas KLEIN

Supervisors:

Prof. Dr. Bernd BISCHL

Janek THOMAS

May 7, 2018

Declaration

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and bibliography.

Munich, May 7, 2018

Niklas Klein

Abstract

We address the problem of hyperparameter optimization for machine learning algorithms. This is a difficult task for two main reasons: On one hand, the function modelled by the machine learning algorithm is usually derivative free (black-box). On the other hand, function evaluations are typically very expensive in terms of their computational time.

The focus of this thesis lies mainly on the hyperparameters of gradient boosting models and neural networks. Both methods contain a large amount of interdependent hyperparameters. This makes the optimization especially difficult.

Simple optimization approaches such as grid or random search generally do not provide satisfying results. We are going to examine more sophisticated strategies like Bayesian Optimization which try to model the underlying black-box function in order to find a better configuration.

A more recent approach is given by Hyperband, which is the eponym of this thesis. In essence, it adaptively allocates resources across a set of randomly sampled configurations based on their performances. We will explore this algorithm in detail and eventually propose a possible way of combining the advantages of model-based optimization and Hyperband. Afterwards, we present our implementation of Hyperband which we based on the R6 class system. This object oriented variant provides a very generic framework which enables the user to easily modify the conventional Hyperband algorithm. We finally present the results of a benchmark experiment where we compare the aforementioned combination of Hyperband and MBO with the original algorithm. Our findings show that for boosting models, both variants perform about equal (for few training iterations, the hybrid optimizer outperforms the conventional Hyperband while for more training iterations, it is the other way around). When considering neural networks, for which Hyperband was initially intended, both Hyperband-based methods outshine the base line model (e.g. a Random Search). Particularly striking is the fact that even though our hybrid approach generated an overhead (through the additional computations resulting from the model-based procedure), it generally needed less computational time than the conventional Hyperband.

Contents

1	Introduction	1
2	The Hyperparameter Optimization Problem	3
2.1	Gradient Boosting	3
2.1.1	Brief Introduction	3
2.1.2	XGBoost and its Hyperparameters	4
2.2	Neural Networks	6
2.2.1	Introduction	6
2.2.2	Hyperparameters of a Neural Network	10
3	Methods for Hyperparameter Optimization	14
3.1	Bayesian Optimization	15
3.1.1	Initial Design	18
3.1.2	Surrogate Models	19
3.1.3	Infill Criteria	22
3.2	Hyperband	24
3.2.1	Successive Halving and the B to n Problem	24
3.2.2	The Hyperband Algorithm	26
3.2.3	Hyperband Example	27
3.3	Combining Hyperband with Model-Based Optimization	30
4	Implementation	34
4.1	A Brief Introduction to R6	34
4.2	The hyperbandr R Package	36
4.2.1	Algorithm Objects	36
4.2.2	Bracket Objects	37
4.2.3	Storage Objects	38
4.3	The hyperbandr Vignette	39
4.3.1	Introductory Guide	39
4.3.2	Optimization of a Convolutional Neural Network with Hyperband	42
4.3.3	Additional Features	49
4.3.4	Extension of Hyperband with Bayesian Optimization	54
4.3.5	Optimization of a Gradient Boosting Model with Hyperband	57
4.3.6	Optimization of a Function with Hyperband	60
5	Benchmark Experiments	63
5.1	XGBoost	63
5.1.1	Experimental Setup	63
5.1.2	Results and Evaluation	66
5.2	Convolutional Neural Networks	76

5.2.1	Experimental Setup	76
5.2.2	Results and Evaluation	78
6	Conclusion and Outlook	83
	Bibliography	88

1 Introduction

Boosting algorithms and neural networks tend to be very complex systems. Many small ingredients interact in a very elusive manner. In addition to this, particularly the neural networks architectures have become much more complicated. For both methods we address the problem of finding a good set of hyperparameters. These cover the actual settings of our algorithm and must be declared outside of the training phase. Hence, we do not directly target the model parameters. The latter ones are optimized during the training phase (e.g. the weights of a convolutional neural network).

One scope of hyperparameters is to control the flexibility of the model. Figure 1 shows the training progress of an arbitrary neural network. After roughly 10 – 15 training iterations, the validation loss (turquoise) begins to increase. It is arguable that the model begins to adapt to the training data too much. But hyperparameters may include many other components, depending on the algorithm at hand.

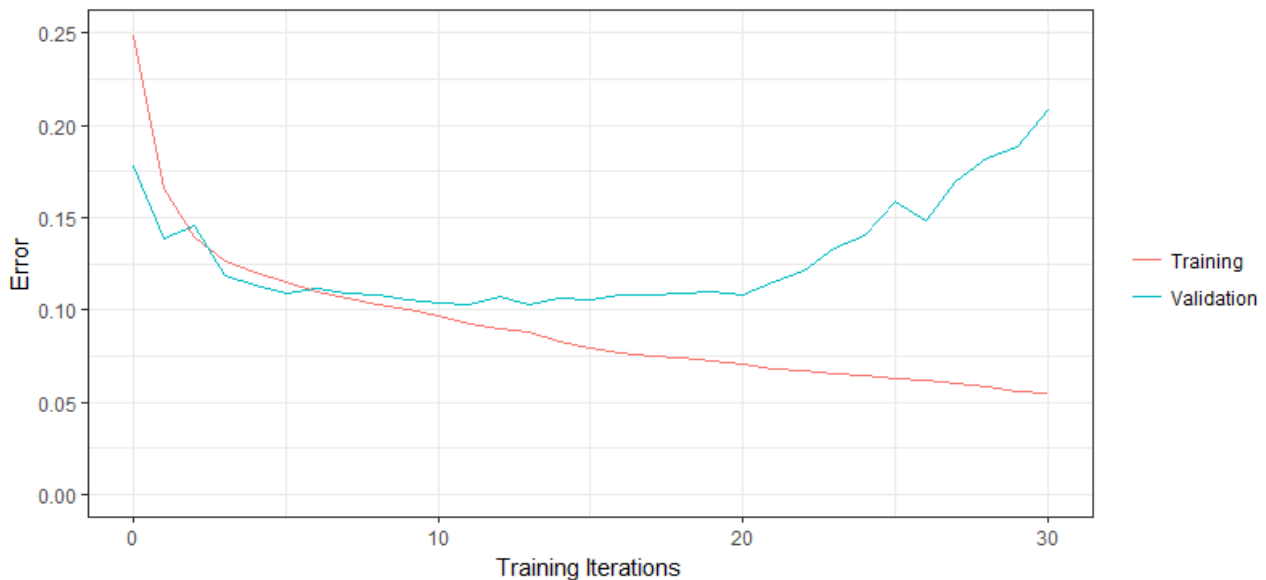


Figure 1: An arbitrary neural network. On the x-axis we see the epochs. That is one forward and one backward pass across the whole training set. By contrast, the y-axis shows the loss of the network. The red curve represents the training and the turquoise one the validation error. After roughly 15 epochs of training, the net seems to begin overfitting the training data.

Machine learning algorithms in general are sometimes referred to as “black boxes”. In science, a black box is a system where one has no knowledge of its internal mechanics (see Figure 2). This could be a computer program, where the user is not allowed to see the code (e.g. due to a closed source program). But also the human brain is considered to be a black box.

So when people use this metaphor for machine learning algorithms, they actually try to paraphrase the vast difficulty of tuning these machines. Severe trouble is mainly caused by the interaction of each of its components and the very costly evaluations. Hence, machine learning algorithms are no black boxes as such. To clarify the reasoning, consider the following thought experiment from Card (2017):

We have some switches and buttons, which represent our hyperparameters. By modifying their state we cause light bulbs to either power or remain disabled. For a small system we could learn the mechanics of our black box by trying all possible input combinations. Unfortunately, as the system grows in size this approach becomes more and more difficult, if not impossible. Even if we obtain access to the wiring diagram, which is our black box, we will not be able to fully explain its behaviour. This is due to the fact that complexity originates primarily from the interaction of ordinary components.

In the context of machine learning, our algorithms are very complicated wiring diagrams which we have access to, and the buttons are the corresponding hyperparameters. Testing a new input combination is usually very expensive in terms of a given budget (e.g. the time).



Figure 2: A black box uses inputs to generate outputs. For the observer it is not possible to comprehend the procedures inside of the black box. He may only see the input and the corresponding output.

2 The Hyperparameter Optimization Problem

In this chapter we introduce both, gradient boosting and neural networks.

After a short introduction to each methods mechanics, we aim towards their individual hyperparameters and the arising difficulties to find a good composition of them.

2.1 Gradient Boosting

In the world of supervised learning, we usually have a vector of input variables x and an output variable y . The goal is to use a labeled training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$ to approximate a function f , which minimizes the empirical risk $\mathcal{R}_{emp} = \sum_{i=1}^n L(y^{(i)}, f(x^{(i)}))$:

$$\hat{f} = \arg \min_f \mathcal{R}_{emp} \quad (1)$$

Gradient boosting (Friedman 2001) is an ensemble method and very successful at this task.

2.1.1 Brief Introduction

We can think of a gradient boosting model as the weighted sum of many weak learners b :

$$\hat{f}(x) = \sum_{m=1}^M \hat{\beta}^{[m]} b(x, \hat{\theta}^{[m]}) \quad (2)$$

These weak learners, solely, are relatively poor performing predictors and in general just slightly above chance. A common choice are small trees or even stumps. So when composing a new boosting model, we gradually add these weak learners to our ensemble. For each weak learner we impose the requirement to improve the overall performance of our model. This is where the eponym of the model class becomes apparent. At each boosting iteration m and for all observations i , we compute the negative gradient of our loss function L :

$$r^{[m](i)} = - \left[\frac{\delta L(y^{(i)}, f(x^{(i)}))}{\delta f(x^{(i)})} \right] \quad (3)$$

The $r^{[m](i)}$ are called pseudo residuals and tell us which way to go in function space in order to reduce the loss. Following up, we fit a weak learner on the negative gradient vector $r^{[m]}$:

$$\hat{\theta}^{[m]} = \arg \min_{\theta} \sum_{n=1}^n (r^{[m](i)} - b(x^{(i)}, \theta))^2 \quad (4)$$

We can think of this concept as fitting a weak learner on the error of the current model. Up next, we find our weighting parameter $\hat{\beta}$ by conducting a line search. To finish the boosting iteration, we add both parts into our ensemble:

$$\hat{f}^{[m]}(x) = \hat{f}^{[m-1]}(x) + \hat{\beta}^{[m]}b(x, \hat{\theta}^{[m]}) \quad (5)$$

In the next iteration, we simply repeat the process by fitting another weak learner on the error of the updated model. This approach is also called forward stagewise additive modelling.

An advantageous aspect of gradient boosting is that it generalizes the idea of boosting. This means that we are allowed to use an arbitrary loss function. Hence, in order to conduct classification, all we have to do is to select it appropriately (e.g. the binomial loss for binary classification problems).

2.1.2 XGBoost and its Hyperparameters

One of the most famous boosting frameworks is XGBoost, an abbreviation for extreme gradient boosting (Chen et al. 2018). It provides a highly parallelized tree boosting implementation and contributed to many winning solutions in various Kaggle challenges. An essential characteristic is its huge variety of regularization parameters. For instance, consider its risk function:

$$\mathcal{R}_{emp}^{[m]} = \sum_{i=1}^n L(y^{(i)}, f^{[m-1]}(x^{(i)}) + b^{[m]}(x^{(i)})) + \underbrace{\gamma J_1(b^{[m]}) + \lambda J_2(b^{[m]}) + \alpha J_3(b^{[m]})}_{regularization} \quad (6)$$

The leading part is essentially the same as we introduced in section 2.1. But behind that we observe three additional terms. The first component γ covers the minimum loss reduction required to make a further split (we can think of the tree depth). Additionally, λ introduces an L_2 and α an L_1 regularization on the models weights.

But these are by no means all hyperparameters we have to deal with. We have to incorporate a learning rate ν , shrinking the step size of each iteration:

$$\hat{f}^{[m]}(x) = \hat{f}^{[m-1]}(x) + \nu \hat{\beta}^{[m]}b(x, \hat{\theta}^{[m]}) \quad (7)$$

This can also be regarded as a regularization in order to prevent overfitting. While γ affects the tree size indirectly, we can also opt to cap it in a rule-based fashion (e.g. a maximum tree depth). The subsample parameter defines the fraction of observations to be randomly sampled for each tree. Inspired by random forests, another regularization technique is feature subsampling (Chen & Guestrin 2016). We can choose to incorporate only a certain subset of columns when growing a tree. Going one step further, we are even able to subsample the features at each split in our trees. Table 1 summarizes the introduced parameter set.

It becomes immediately apparent that finding the right configuration is very hard. Chapter 3 will introduce various techniques to tackle this issue.

Table 1: XGBoost feasible hyperparameters. Most of them aim towards regularization. The range column indicates each hyperparameters possible region for their values.

parameter	range	description
γ	$[0, \infty]$	minimum loss reduction required to make a split
λ		L_2 regularization term on the weights
α		L_1 regularization term on the weights
max_depth		The maximum allowed depth of each tree
subsample	$(0, 1]$	fraction of randomly sampled instances for each tree
colsample_bytree	$(0, 1]$	fraction of randomly sampled features for each tree
colsample_bylevel	$(0, 1]$	fraction of randomly sampled features for each split

2.2 Neural Networks

As of 2018, neural networks play a vital role in many fields. For sophisticated projects, such as the realization of autonomous cars, they can act in a supportive fashion. This includes subtasks for the perception and localization of the car as well as the control over the drivers state (Fridman 2018). Neural networks also find use for mundane tasks, like speech recognition. In order to predict the input sequence of phonetic states from audio data, Amazons virtual assistant “Alexa” utilizes a distributed variant of recurrent neural networks (Strom 2015). But also expert problems, like the analysis of medical images, are being tackled successfully by deep neural networks (Qayyum et al. 2017).

2.2.1 Introduction

Neural networks in general integrate the process of feature engineering into their modeling pipeline. The closer past has shown that this works particularly well for certain types of data. Judging by the results of the annual ImageNet contest (Russakovsky et al. 2015), convolutional neural networks represent the undisputed state of the art for image processing. This covers for instance object and speech recognition, but also segmentation exercises.

Convolutional neural networks, from now on abbreviated CNNs, are inspired by the visual cortex of mammals. Figure 3 describes the fundamental idea. Initially, a stage named V1 extracts so called low level features. These include edges, lines or color gradients. Subsequent stages assemble these low level features to increasingly compounded structures. We can think of ordinary Lego bricks in the first stage. These are plugged together to form shapes as depicted by stage V2. Finally, these shapes are assembled to yield “complete” objects as depicted by stage V4. Eventually, the human brain can recognize such objects and assign them to a class (e.g. a face).

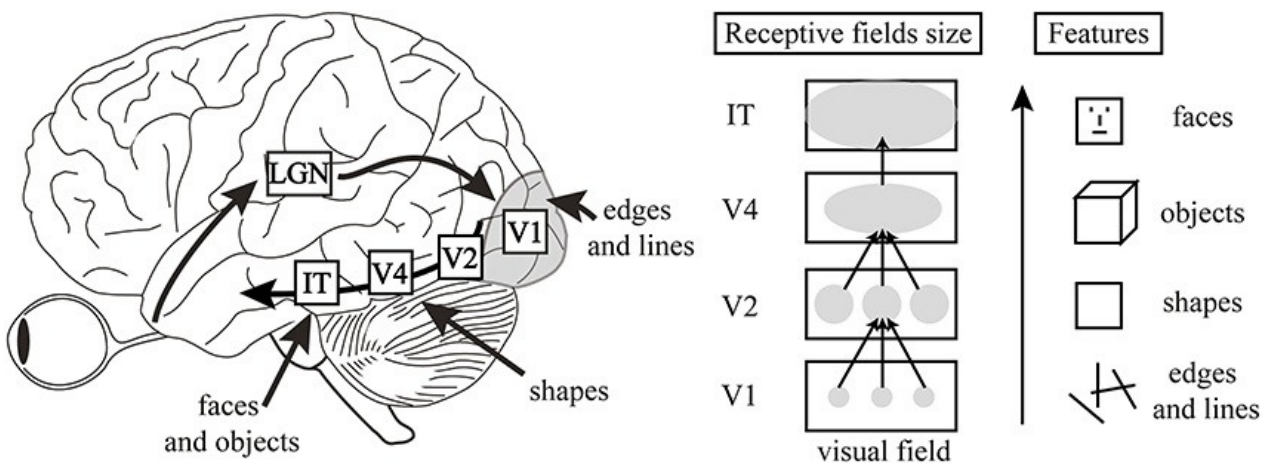


Figure 3: Visual cortex and feature extraction (Herzog & Clarke 2014). The image describes how the visual cortex of a human works. The assembly of low level features (e.g. edges, lines or color gradients) leads to complex high level features such as faces.

To cast this idea into a model, we have to become acquainted with the name giver of the model class, which is the discrete convolution. Consider Figure 4a. The left side shows a simple

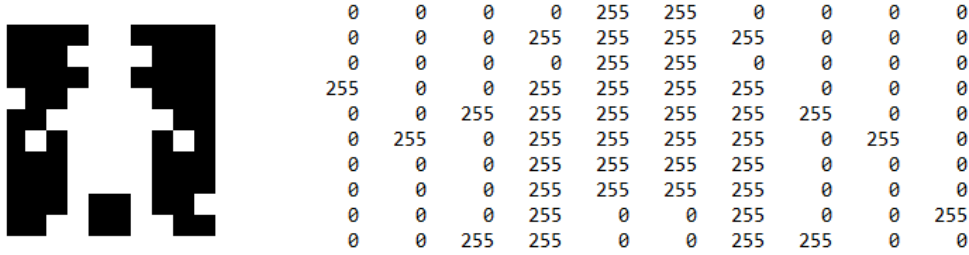
black-and-white image. In contrast, the right side represents the associated pixel entries (i.e. what a computer “sees”). We deduce zeroes representing the color black, 255 the color white and everything in between the gray levels.

Now suppose we would like to detect vertical edges in that image. One way to achieve this is by applying a filter called Sobel-Operator (Sobel 1968). Figure 4b shows one step of the required procedure. We position the Sobel-Operator on the red framed location of the image and compute the dot product. That is:

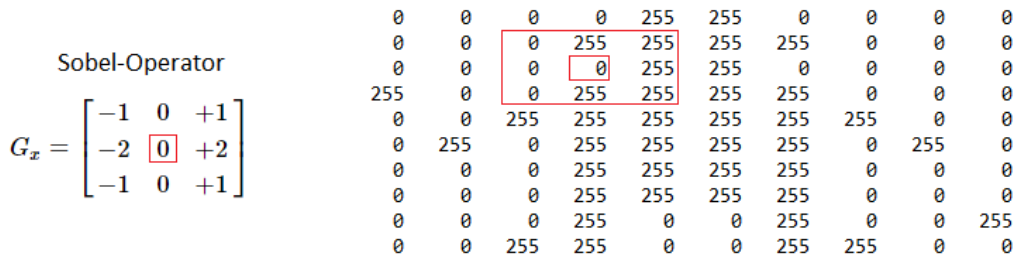
$$\begin{aligned} S_{(i,j)} = (I \star G_x)_{(i,j)} = & -1 \cdot 0 + 0 \cdot 255 + 1 \cdot 255 \\ & -2 \cdot 0 + 0 \cdot 0 + 2 \cdot 255 \\ & -1 \cdot 0 + 0 \cdot 255 + 1 \cdot 255 \end{aligned} \quad (8)$$

where $S_{i,j}$ indicates the output at row i and column j . The \star implies convolution between the input image I and the filter G_x . After applying the filter to every feasible location of our input image (Figure 4c), we have to normalize the pixel entries, as can be seen in Figure 4d. The resulting output array is called feature map.

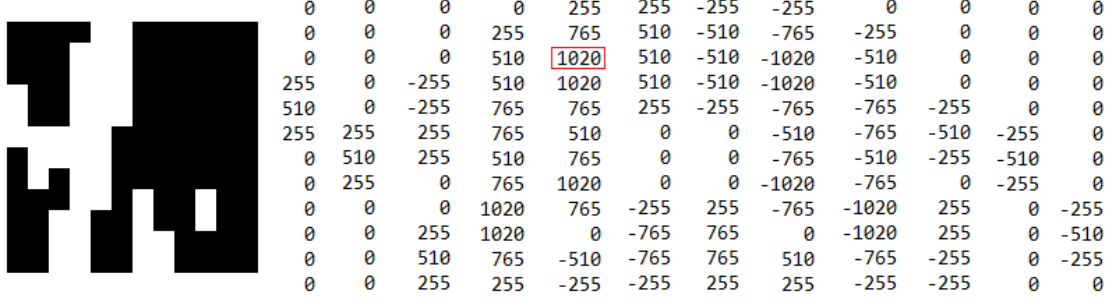
What the Sobel-Operator actually did was investigating the neighborhood of each central pixel. Technically, this equals the computation of an approximation of the gradient.



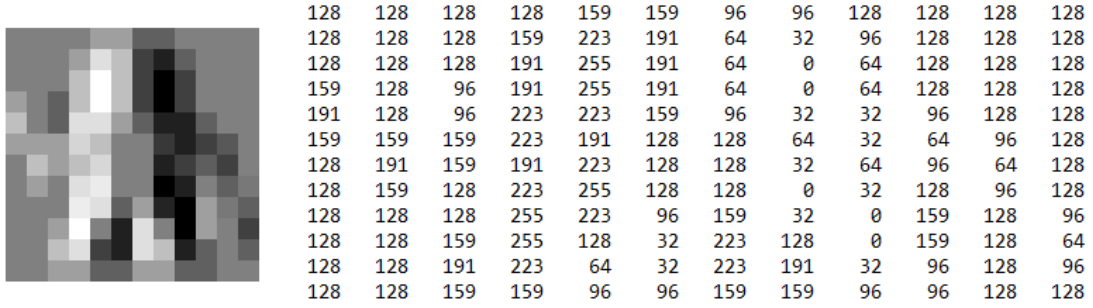
(a) How to represent a digital image: ordinary black-and-white image on the left, pixel entries on the right. Zeros indicate the color black, 255 the color white and everything in in between the gray levels.



(b) Exemplary application of the Sobel-Operator on an arbitrary location of our input image. We simply compute the dot product to obtain the convolved value: $S_{i,j} = 1 \cdot 255 + 2 \cdot 255 + 1 \cdot 255 = 1020$ (more details of the computation can be seen in Equation 8).



(c) Results after applying the Sobel-Operator to every possible location in our input image. Framed in red: the outcome of the application of the Sobel-Operator to the area shown in Figure 4b.



(d) Normalized pixel values to reveal the vertical edges in the input image. The corresponding image is called feature map.

Figure 4: Applying the Sobel-Operator on a black-and-white image in order to detect vertical edges.

Figure 5a summarizes what happened: a predefined filter was used to convolve the input image into a feature map. What a convolutional neural network does is very similar. One crucial difference arises in the fashion filter entries, which are our model parameters, are selected. In fact, a CNN initializes them by a random rule (see Figure 5b).

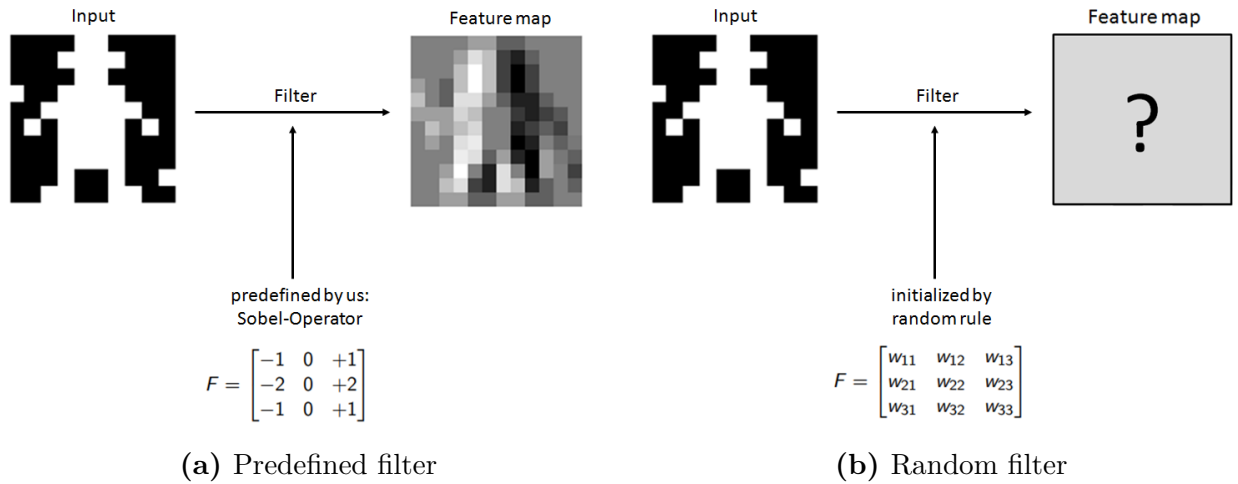


Figure 5: Convolutional neural networks use randomly initialized filters to output feature maps. An optimization algorithm is used to iteratively update the parameters of the filter.

To construct a model of this idea, we use filters for automatic feature extraction and dense layers to associate them. Suppose we would like to recognize objects from the CIFAR-10 dataset (Krizhevsky 2009). It consists of 60.000 32×32 color images, evenly distributed over 10 classes. Those classes are cars, trucks, airplanes, birds and other animals. A potential architecture to do this could look like it is shown in Figure 6. We have some feature maps in our first layer. Each of them is constructed by an individual, randomly initialized filter.

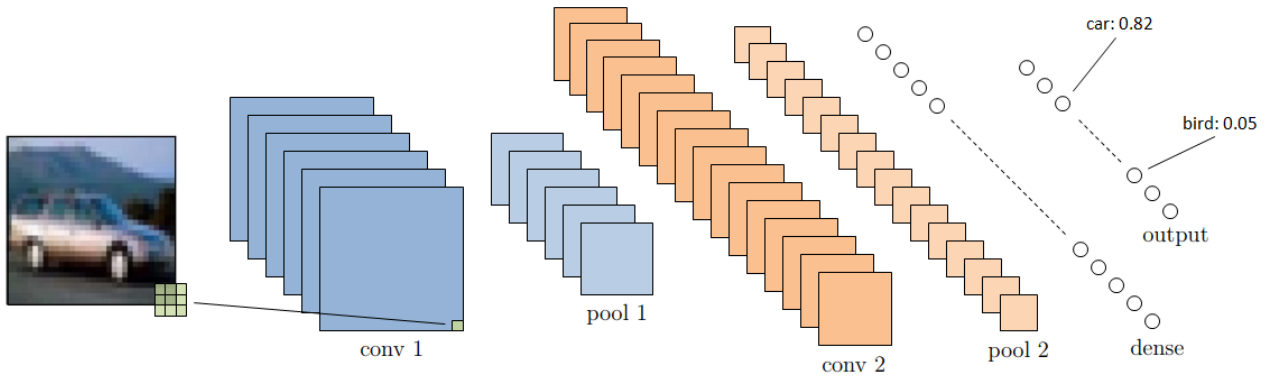


Figure 6: Architecture of a small convolutional neural network. The model has two convolution and corresponding pooling layers, followed by one dense and one output layer.

Subsequently we apply an operation called pooling to reduce the size of our feature maps. As a consequence, we get rid of a lot of data. One advantage is that our computation is less expensive. While doing so, we try to retain as much information as possible. One typical approach for this is called max pooling and depicted by Figure 7. We use the common filter size 2 as well as the standard step size of 2. That means that for each 2×2 block we extract its maximum value and eventually quarter the size of our feature map.

Our model architecture of Figure 6 repeats the whole procedure one more time. In the second round we use the features of pooling layer 1 to produce the second convolution layer. That is the part where we plug our Lego bricks together in order to build more sophisticated structures.

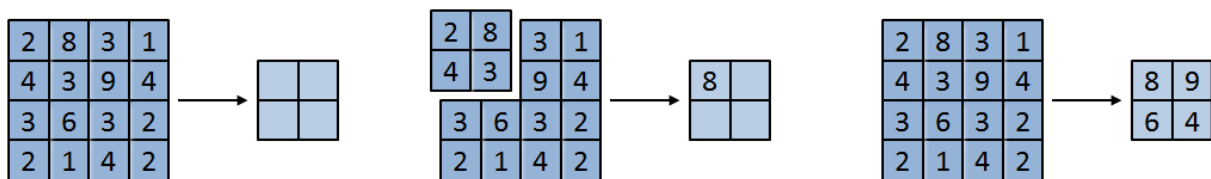


Figure 7: Max pooling procedure. The blue array represents a feature map. We choose a 2×2 filter and a step size of 2. For each 2×2 block, we extract the maximum value. At the first location we obtain $\max\{2, 8, 4, 3\} = 8$. The two by two rectangle represents the pooled feature map.

Finally, we feed the features which are contained in pooling layer 2 into a dense layer. This is our actual predictor. Each time we propagate a batch of images through the network, we compare the results of our predictor with the ground truth. For each of our parameters and quite similar to chapter 2.1.1, we compute the negative gradient of a loss function. This will tell us in which way we have to adjust each parameter of our model, in order to reduce the

loss. We call this propagating the error backwards through the net to measure each parameter's contribution to it. To obtain a good model, we have to repeat this step many times. Thus, fitting a neural networks is an iterative procedure. One forward pass and one backward pass across the whole training set is called epoch. Generally, we repeat the whole technique for multiple epochs.

2.2.2 Hyperparameters of a Neural Network

When we decide to address a problem with a neural network, the first challenge is to find a suitable architecture. Once we found a basic framework to operate with, the next problem appears:

Neural networks have an inconceivable amount of hyperparameters, which makes tuning very hard. In order to ease the problem a little bit we settle on the architecture of Figure 6. That means that we fixate some hyperparameters, such as filter sizes or the amount of feature maps and neurons in all layers.

A feasible architecture is demonstrated by Table 2. Since CIFAR-10 contains only color images, we have to incorporate three channels (red, green and blue) for the filters of our first conv layer. As a consequence, each filter in layer one has $3 \times 3 \times 3 = 27$ parameters. Including the bias, the first layer has a total of $3^3 \times 8 + 8 = 224$ model parameters. The whole architecture has 66.634, while most of them result from the dense layer with 64 neurons ($1.024 \times 64 + 64 = 65.600$)

Table 2: Model architecture of Figure 6. Since we're dealing with color images, the filter of our first layer has to incorporate the channels of the RGB color model. Hence, each 3×3 filter of layer one has $3^3 = 27$ parameters. For convolution filters we chose padding and a step size of 1. Thus, both conv layers will conserve its inputs height and width. The pooling filters have a dimension of 2×2 as well as a step size of 2.

layer	filter/neurons	feature maps	model parameters	current dimension
conv 1	$3 \times 3 \times 3$	8	224	32×32
pool 1	2×2			16×16
conv 2	3×3	16	160	16×16
pool 2	2×2			8×8
flatten				1×1.024
dense	120		65.600	
output	10		1.210	
total			66.634	

Before we can utilize this architecture, some obligatory hyperparameters must be set.

We described that after processing a batch of data, we need to calculate the error contribution

of each weight. This is done by an enveloping optimization algorithm. Figure 8 shows that beside stochastic gradient descent, a lot more options to choose from are available. While the three shown examples all share the “sub-hyperparameters” learning rate and learning rate decay, they also introduce some individual ones. For example, when we opt for sgd, we also have to find a good value for momentum and decide whether we want to apply nesterov or not.

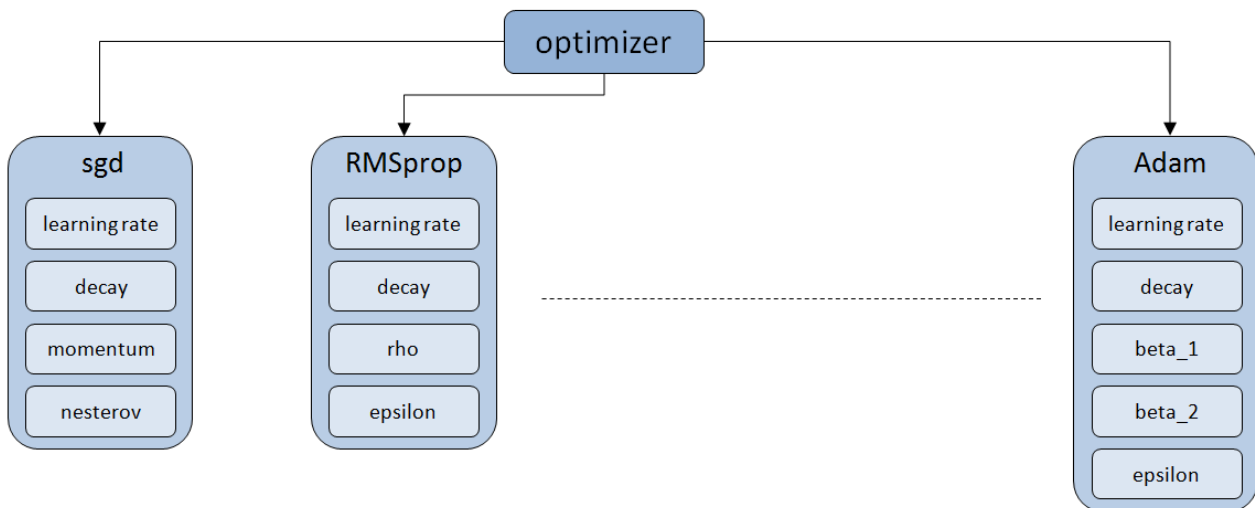


Figure 8: The optimizers for a neural network. All shown examples share some “sub-hyperparameters”. For instance the learning rate or the learning rate decay. But they also introduce individual ones, which makes the search for the right choice even more difficult.

Another global hyperparameter is the batch size. If we set it too small, convergence will take really long. Choosing it too high, the performance of our model will very likely suffer (Keskar et al. 2016). There is also a strong interaction between the learning rate and the batch size. A larger batch size requires a smaller learning rate. Smith et al. (2017) propose adaptive batch sizes. They argue that after some epochs with small batches, we have very likely skipped bad local minima and eventually converge into a good point faster.

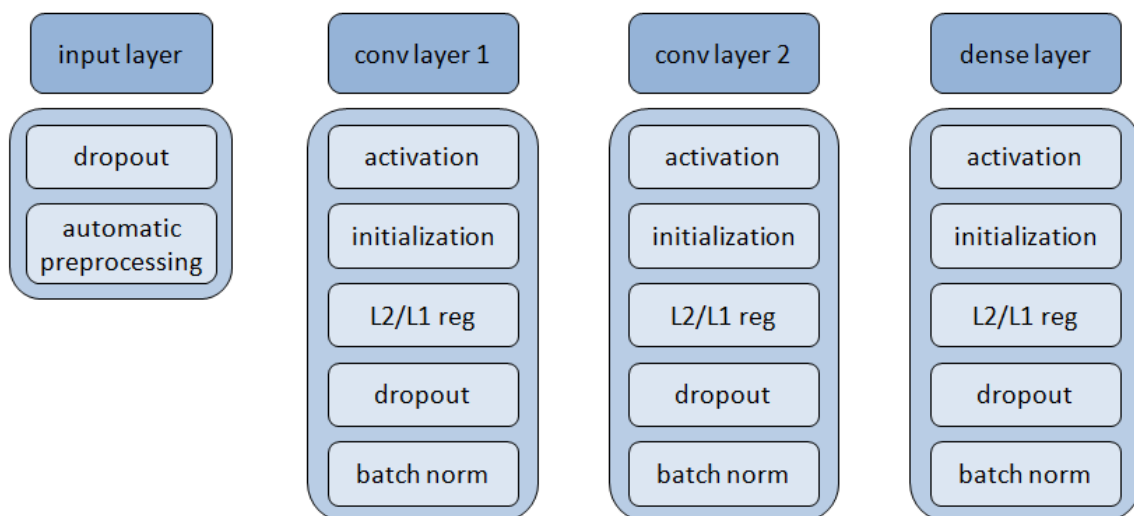


Figure 9: Layers and hyperparameters. Particularly appealing when dealing with image data, one could utilize dataset augmentation on the input as automatic preprocessing. For conv and dense layers, a crucial hyperparameter is the activation function. Furthermore, a wide range of regularization parameters are available.

Each individual layer represents another building block full of hyperparameters. That includes the input as well. We could opt to apply live dataset augmentation within our model pipeline. For images one can think of flipping each instance by a given degree as well as mirroring each of them. Doing that, we can easily increase our train set by a magnitude of 8.

Zhong et al. (2017) even apply regularization on the input by randomly erasing parts of each image (see Figure 10).

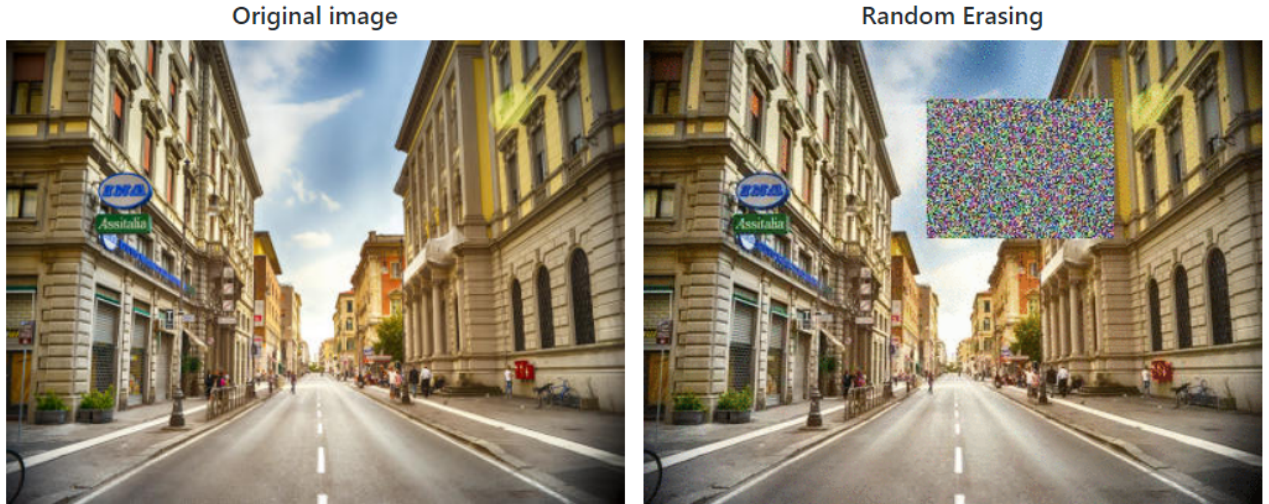


Figure 10: Randomly erasing parts of input images in order to increase regularization (Zhong et al. 2017).

As we progress to the actual layers, one indispensable hyperparameter is a set of activation functions. They are attached at the end of each layer and supply the model with non-linearity. While the standard ReLU ($\max(0, x)$) is usually a good choice, more sophisticated mutations are operational. For instance the LeakyReLU, which aims to fix the dying ReLU problem. The latter one means that a ReLU outputs 0 for any input and since the gradient at that state is also always 0, it is very unlikely that it can recover on its own. LeakyReLUs try to overcome this issue by assigning a very small negative slope for values $x < 0$ (Maas et al. 2013).

When we start the training process, we initialize all model parameters by a certain rule. This weight initialization can have a significant impact on the networks convergence rate and thus on the overall quality of a model. The neural network library Keras (Chollet 2015) for example provides over 15 variants. Those include conventional sampling from a normal distribution but also advanced techniques that incorporate the structure of the input data. Xavier initialization for instance tries to match the variance of the outputs of a layer with the variance of its inputs. According to their inventors, this is working particularly well for layers with sigmoidal activation functions (Glorot & Bengio 2010). He et al. (2015) captured this idea and customized a version called “He normal” for the ReLU family.

Each layer of a neural network can be furnished with various regularization techniques. One of the most famous is called Dropout (Srivastava et al. 2014). For each batch that we propagate through the network, a preset fraction of neurons will be randomly deactivated. It is believed that this prevents hidden units from learning every detail of the data and thus to memorize it. Alternatively or even in addition to dropout, we could employ an L1 or L2 regularization (in

deep learning, the latter one is commonly known as weight decay).

A recent advance in deep learning is called batch normalization (Ioffe & Szegedy 2015). The idea is to normalize the distribution of each input unit of a layer across the current batch. Since the inputs for the next layer are based on all instances of the current batch, it does also introduce regularization, as well as an interaction with the learning rate. Their originators claim that batch normalization enables the usage of higher learner rates which will speed up training time. While we covered a striking amount of hyperparameters, it still represents only a small fraction of possibilities when dealing with neural networks. Many more subject-specific options like gradient clipping or the choice of custom tailored right loss functions can be taken into account.

3 Methods for Hyperparameter Optimization

The last chapter introduced gradient boosting, and technically speaking, convolutional neural networks. Both methods provide a large variety of hyperparameters. This makes the search for a good composition very tedious. People quite often do not have sufficient experience to “guess” good parameters. They typically rely on default or recommended values. Unfortunately, these are not always suitable for the problem at hand.

This issue is aggravated by the fact that literature on good tuning techniques is very under-represented (Bischl 2017). Brute force methods like grid search are easy to implement but not really helpful. The combinatorial explosion makes it very expensive. In addition to this, it will very likely waste a lot of time on irrelevant areas. Likewise, as it relies purely on luck, random search isn’t a reliable choice either. Yet, it often serves as the standard baseline model.

The upcoming subchapter introduces a much more sophisticated technique which is called Bayesian or model-based optimization. Its central idea is to propose new configurations in a sequential fashion.

Figure 11 hints at each method’s characteristics. Here we try to find a good combination of momentum and input dropout for a neural network. Suppose we are limited by a temporal restriction and thus can only train for 10 epochs. Obviously, we want a high momentum in combination with a low to medium dropout value. The grid search spends a lot of resources in bad areas. Random search does basically the same. MBO however quickly zeros itself in on a promising area. Both grid search and random search perform 25 model evaluations. Their lowest mean missclassification errors (mmce) are 34.53% and 39.25% respectively. Despite conducting only 18 model evaluations, MBO manages to get a mmce of 11.46%.

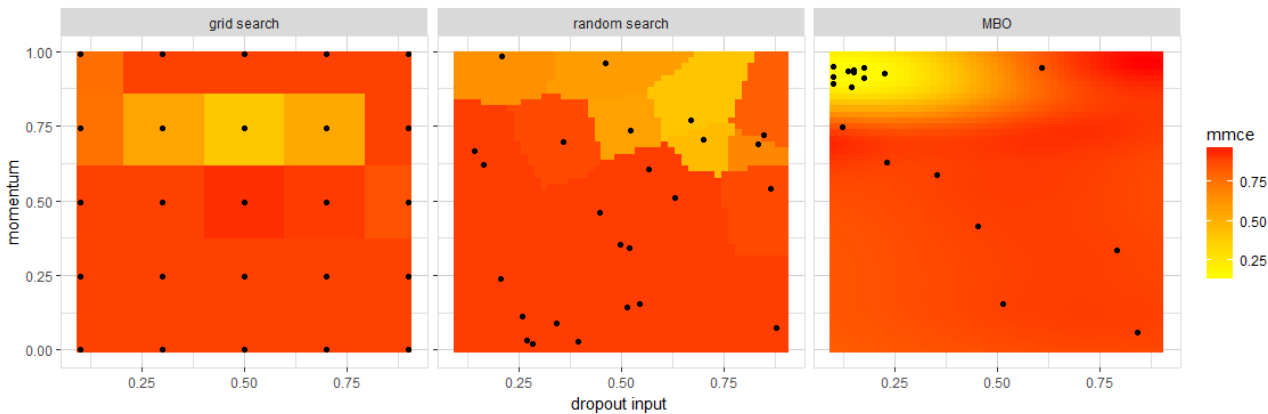


Figure 11: Grid search, random search and MBO. We try to find a good combination of momentum and input dropout. Both grid search and random search waste a lot of resources on irrelevant areas. MBO needs fewer evaluations to find a better configuration. Its lowest mmce is at 11.46%, while random search and grid search only manage to get an error of 34.53% and 39.25% respectively.

Chapter 3.2 presents a completely different approach to hyperparameter optimization, called Hyperband. Especially for neural networks, the relatively new algorithm provides state of the art results. In the end we propose a general strategy to combine both methods: model-based optimization and Hyperband.

3.1 Bayesian Optimization

Assume we would like to minimize an unknown function f . This means in particular that we are not able to compute derivatives. The only thing we can do is to observe the function's output y for an arbitrary set of input parameters $x \in \mathbb{X}$:

$$y = f(x)$$

$$f : \mathbb{X} \rightarrow \mathbb{R}$$

To minimize the function, our task is to find the best input combination:

$$x^* = \arg \min_{x \in \mathbb{X}} f(x)$$

Now suppose the function is very expensive to evaluate. We can think of a high monetary cost for each trial of an engineering problem. Another paradigm could be a long-term study in order to test the effect a new drug. For machine learning models, the costliness arises from the time it will take to train one model. Therefore, we cannot simply “brute force” a good result by performing an endless amount of evaluations.

Bayesian optimization is a sequential and derivative free approach to tackle these kinds of problems. The central idea is to use regression to approximate the unknown target function. Algorithm 1 briefly describes the general strategy.

We begin to draw $|n_{init}|$ configurations from the parameter space \mathbb{X} . Following up, each one of them is being evaluated: $y^{(d)} = f(x^{(d)})$. The tuples of configuration and corresponding output are called the initial design:

$$\begin{aligned} & (x^{(1)}, y^{(1)}) \\ & (x^{(2)}, y^{(2)}) \\ & \vdots \\ & (x^{(n_{init})}, y^{(n_{init})}) \end{aligned}$$

In the next step, we fit a regression model on this design. We call it the surrogate model and it essentially tries to interpolate the unknown function. This model must also include predictions for the uncertainty at each location.

A so called infill criterion utilizes this information to propose a new interesting configuration. In a nutshell, it governs the trade-off between exploitation and exploration. To put it another way, if we inspect regions with high variance, we conduct exploration. Going into areas with good estimated values on the other hand means exploitation.

The final step is to evaluate the new configuration and add it to the design. We then repeat the procedure until we exhaust a certain amount of budget (e.g. time).

Algorithm 1 Bayesian optimization pseudo code

- 1: **Input:** budget and parameter space \mathbb{X}
 - 2: Sample $|n_{init}|$ configurations $x^{(d)}$ from \mathbb{X} .
 - 3: Evaluate each configuration: $y^{(d)} = f(x^{(d)})$
We call the tuples $(y^{(d)}, x^{(d)})$, $\forall d \in (1, \dots, n_{init})$ the *initial design*.
 - 4: **while** budget not exhausted **do**
 - 5: Fit a regression model on the *current design* to predict $\hat{f}(x)$.
 We call this the *surrogate model*.
 - 6: Propose a new configuration $x^{(d+1)}$ via an infill criterion.
 This governs the trade-off between exploitation and exploration.
 - 7: Evaluate the new configuration and add it to the *design*
 - 8: **end while**
 - 9: **return** best configuration
-

In order to obtain a better understanding of this intuition, let us consider a graphic example. Suppose we would like to find the minimum of Equation 9:

$$f(x) = x^2 + \sin(1.1 \cdot \pi \cdot x) \quad (9)$$

While in practice this would be unknown, the black solid line in each top plot of Figure 12a, b, c and d shows the functions true course.

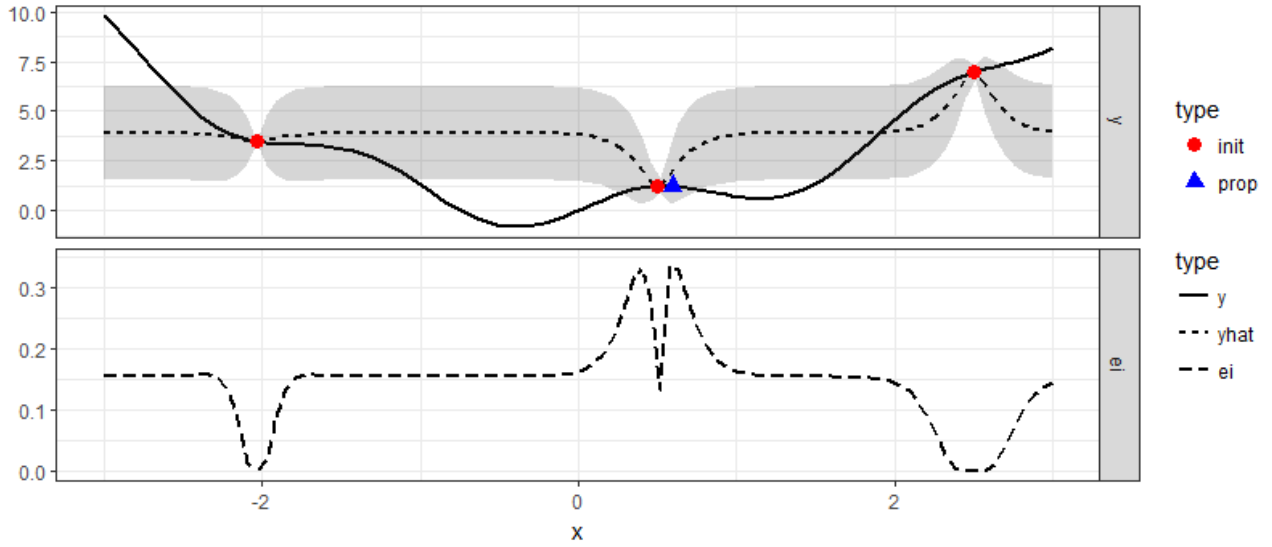
We choose $n_{init} = 3$ for our initial design. The evaluated configurations are represented by the three red points in Figure 12a. Based on them, the dashed line in the upper plot of Figure 12a represents the current prediction of the surrogate model.

Highlighted in grey we also see the prediction of the standard error. These are the areas we mean when speaking of exploration. The bottom plot of Figure 12a show the infill criterion (here we use the expected improvement “ei”, which will be discussed in more detail in section 3.1.3). Its maximum value tells us which configuration we should try next (the blue triangle in the upper plot).

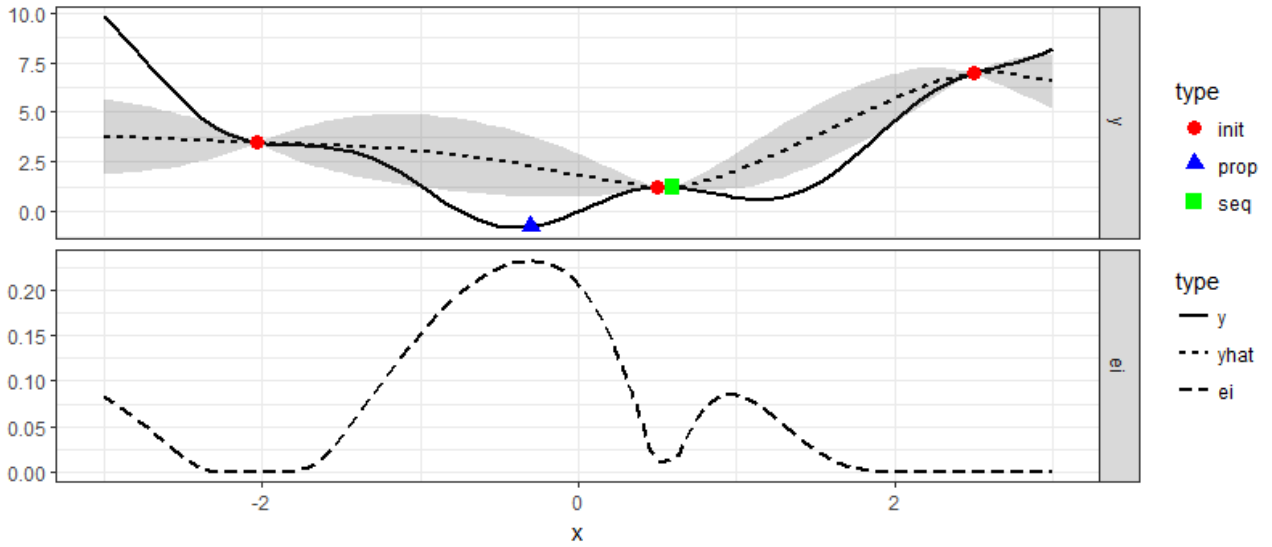
Figure 12b shows the updated surrogate model. A green square depicts the new configuration which is now also part of our design. In this iteration, the infill criterion leads us almost to the global optimum of the function.

The third iteration is portrayed by Figure 12c. We can see that the infill criterion now proposes a point which is slightly worse than the previous evaluation.

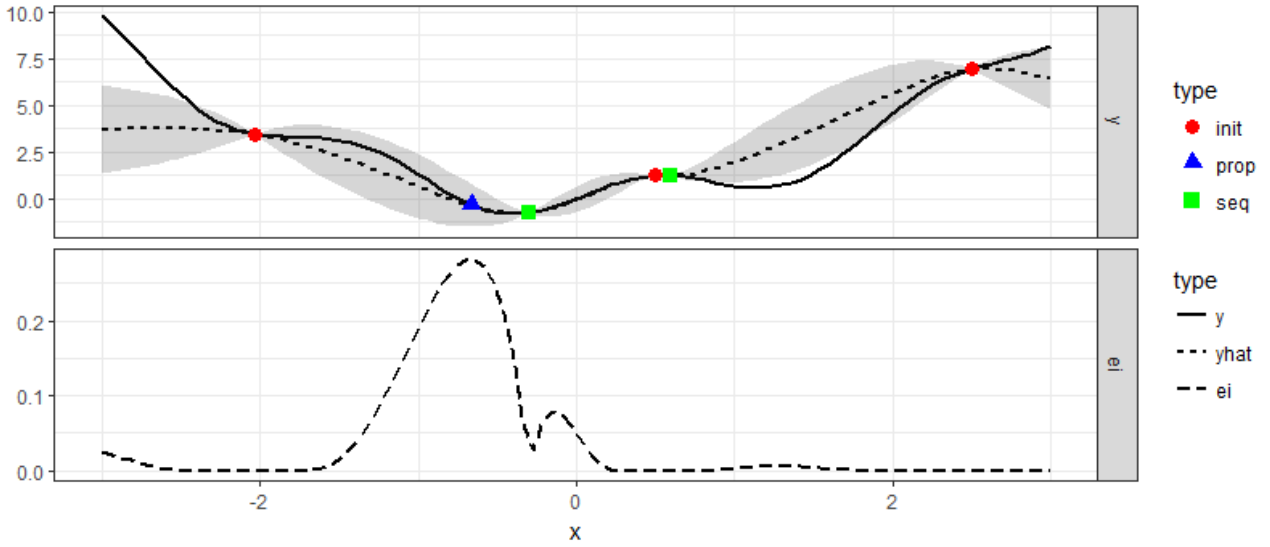
Figure 12d demonstrates the situation after a total of 5 iterations. Eventually, we end up very close or even at the global minimum.



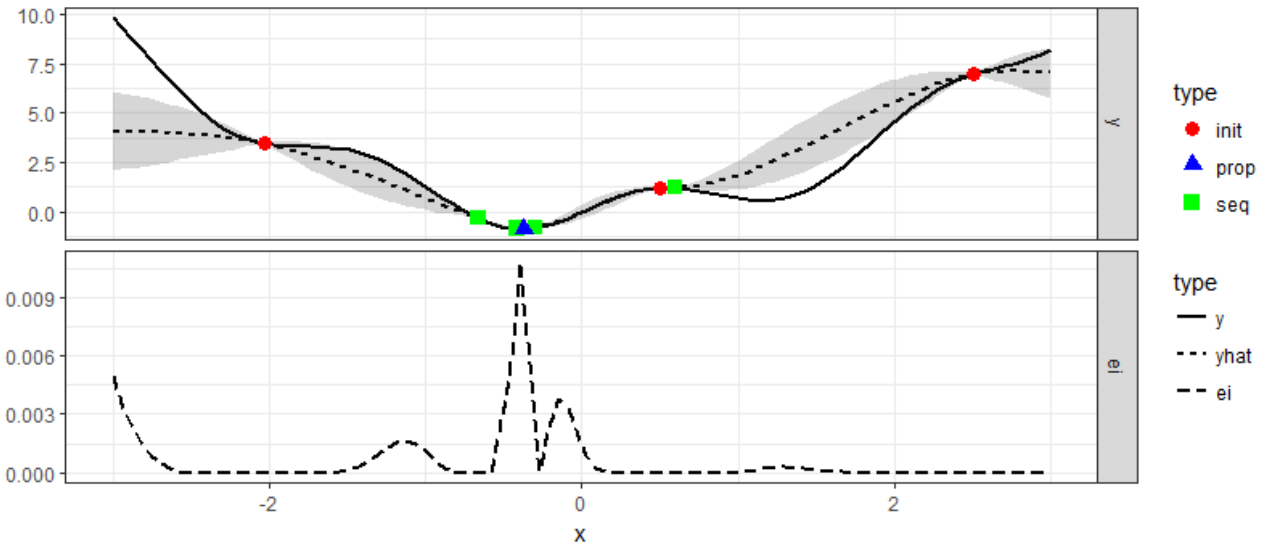
(a) The black solid line in the top plot shows the function of Equation 9 (which we normally know nothing about). Three red dots represent the initial design. A dashed line displays the current prediction of our surrogate model. Shaded in grey we see the standard deviation for each point, which was also estimated by the surrogate model. In the bottom plot, another dashed line represents the infill criterion (here we use the expected improvement, “ei”). It proposes to evaluate the x-value of its maximum next. This configuration is indicated by the blue triangle in the upper plot. Therefore, we decide to exploit instead of explore.



(b) After the first iteration, we incorporate the new point (the green square) into our design. The dashed line in the upper plot shows the new prediction of the surrogate model. According to the infill criterion, we should now evaluate the blue triangle.



(c) After the second iteration, the infill criterion proposes a new configuration, which is slightly worse than the previous one.



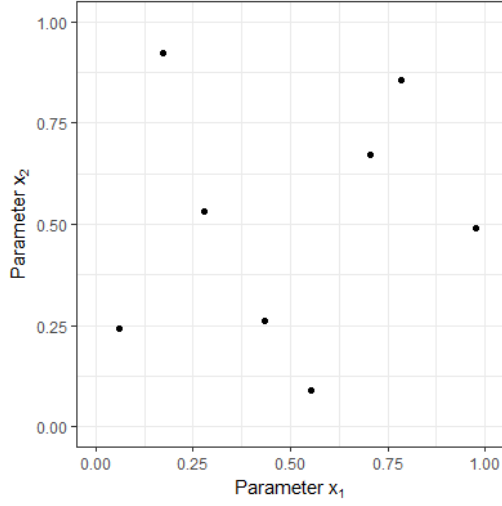
(d) The fifth iteration shows that we end up very close to the global minimum.

Figure 12: Bayesian optimization of Equation 9. We see the development of the surrogate model (black solid line) over 5 iterations. Eventually, we find the global optimum. Both, the computations and plots were realized with the mlrMBO package (Bischl et al. 2017).

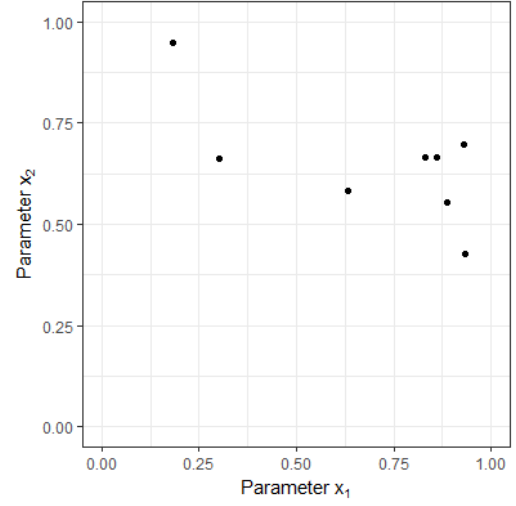
3.1.1 Initial Design

The first step in Bayesian optimization entails the creation of an initial design. This initial design represents the foundation for our surrogate model. Two common alternatives to obtain a set of configurations include random and latin hypercube sampling (LHS).

LHS will maximize the minimum distance between the configurations. Figure 13 shows a simple example for a problem with two parameters. In Figure 13b we can see that random sampling may result in very akin configurations. Yet in certain situations, unequal distances might even be advantageous (Morar et al. 2017).



(a) Sampling with LHS



(b) Sampling completely at random

Figure 13: LHS vs random sampling. LHS will maximize the minimum distance between each configuration. This ensures more exploration but does not necessarily lead to a better result.

3.1.2 Surrogate Models

The key task of the surrogate model is to interpolate the unknown function $f(x)$. Choosing a model depends mainly on the domain of the input variables. A Gaussian process for instance can only handle numeric values. If we incorporate categorical variables, one possible solution could be supplied by random forests. Both of these methods provide a native uncertainty estimation. This is crucial if we want to use an infill criterion.

Gaussian process

In general, a Gaussian process (GP) is a collection of random variables. Every finite subset of these random variables has a multivariate normal distribution (Snoek et al. 2012). We introduce the GP as the distribution over functions:

$$f \sim GP(m, K) \quad (10)$$

where $m : \mathbb{X} \rightarrow \mathbb{R}$ represents the mean function:

$$m(x) = \mathbb{E}[f(x)] \quad (11)$$

and $K : \mathbb{X}^2 \rightarrow \mathbb{R}$ the covariance function:

$$K(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x')))] \quad (12)$$

Now assume we would like to know the function values $f(x)$ for some data points $x = (x^{(1)}, \dots, x^{(n)})$. Since $f(x) = (f(x^{(1)}), \dots, f(x^{(n)}))$ is a subset of random variables, it has the joint distribution

$$f(x) \sim \mathcal{N}(m(x), K(x, x'))$$

So basically the entire Gaussian process is defined by its mean and covariance function. In order to infer a new point, we condition the unknown function value on the known ones:

$$\mathbb{P}(f(x^{(n+1)})|x^{(n+1)}, f(x^{(1)}), x^{(1)}, \dots, f(x^{(n)}), x^{(n)})$$

For each new point the GP returns the mean and the variance of a normal distribution over all possible values of f at x . If there are no “nearby” known points, the mean function will dominate the result. Crucial for this is the choice of the covariance function as it essentially acts as a kernel to measure the similarity of two points.

Recall the problem of Equation 9 and the initial design shown in the upper plot of Figure 12a. Figure 14 compares the standard Gaussian covariance function:

$$\begin{aligned} k(h) &= \exp\left(-\frac{1}{2\theta^2}\|h\|^2\right) \\ &= \exp\left(-\frac{1}{2\theta^2}\|x, x'\|^2\right) \end{aligned}$$

with the Matérn_{3/2} (which was also used for the surrogate model of Figure 12):

$$k(h) = \left(1 + \frac{\sqrt{3|h|}}{l}\right) \exp\left(-\frac{\sqrt{3|h|}}{l}\right)$$

The dashed line represents the Gaussian Kernel and the shaded turquoise area its uncertainty estimation. It appears to be smoother than the Matérn_{3/2} (the solid black line and the red area respectively). This might be more suitable for problems with little prior knowledge.

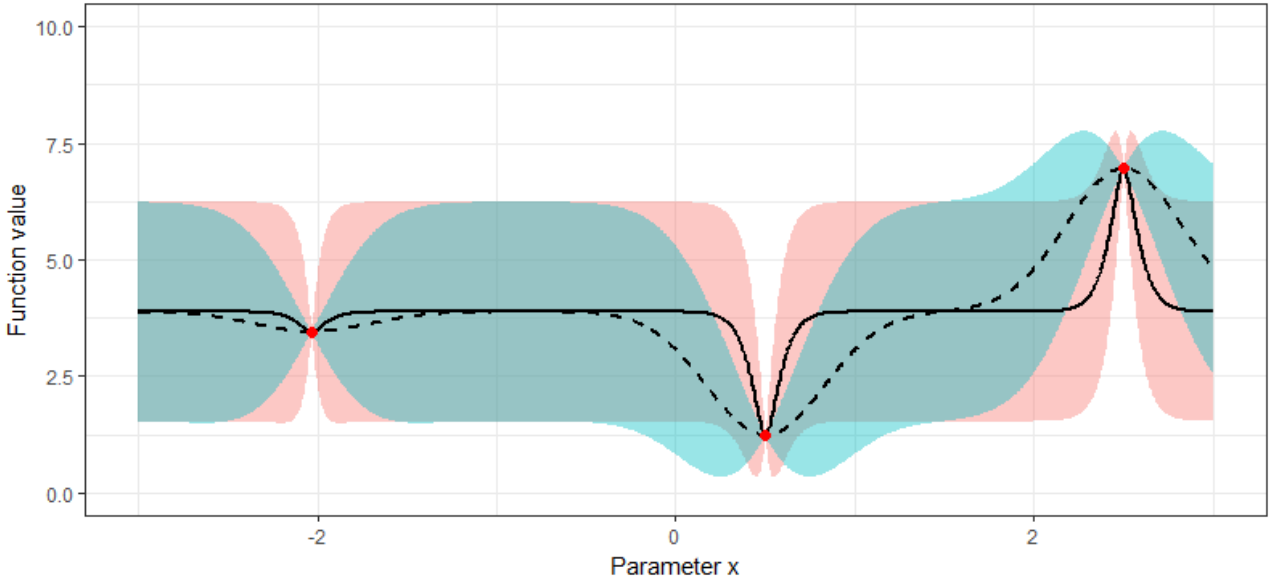


Figure 14: Gaussian covariance kernel vs Matérn_{3/2}. The dashed line represents the mean function of the Gaussian kernel. In turquoise, we see its uncertainty estimation. The Matérn_{3/2} is depicted by the solid line and in red we can see its estimation of the standard error. In the Matérn_{3/2}, the mean function dominates very early. Thus it is a very flexible and general kernel and in addition to that does not assume quite as much smoothness as the Gaussian kernel. Therefore, it might be a good choice when dealing with very little prior knowledge. The plot was generated with the DiceKriging package (Roustant et al. 2012), which uses “kriging”. This is a variant of Gaussian process regression with slightly different assumptions about the mean (Jones et al. 1998).

Random forests

For mixed parameter spaces which include categorical variables we need another model class. A suitable choice is given by random forests as they naturally handle this kind of data. Random forests are an ensemble learning method, primarily applied on classification and regression problems (Breiman 2001).

To fully understand the idea behind random forests, we have to take a step back and acquaint ourselves with Classification and Regression Trees (Breiman et al. 1984). Such a tree divides the feature space \mathbb{X} into M rectangles R and fits a simple model to each of them (e.g. a constant):

$$f(x) = \sum_{m=1}^M c_m \mathbb{1}(x \in R_m)$$

Figure 15 shows the results of three different regression trees. Once again, we use the function of Equation 9 for our demonstration. To compute the model for the left plot, we used 20 equidistant function evaluations (points). The trees of the plot in the middle and right plot had 30 and 50 observations respectively at their disposal. Several stopping criteria, such as the minimal number of observations per node, prevent the tree from overfitting.

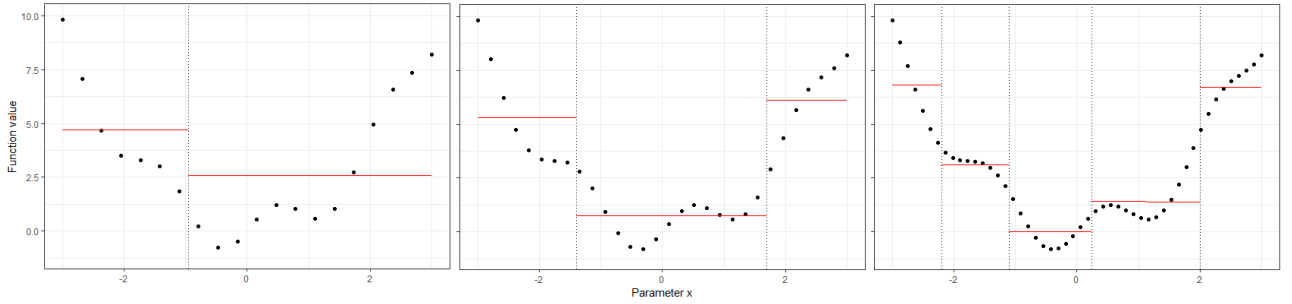


Figure 15: Regression tree on the sinus function of Equation 9. For the left tree we used 20 equidistant function evaluations (points). The tree of the plot in the middle used 30 and the right 50 points respectively. Each classification tree was created with the rpart package (Therneau & Atkinson 2018).

CART is a very simple algorithm. One drawback is its vulnerability to minor changes in the data. If we apply a trained CART model on new data, we will likely experience very high variance. This inevitably means that the predictive performance of a single tree is very poor (Hastie et al. 2009a).

Leo Breiman (1996) proposes bootstrap aggregation (abbreviated bagging) in order to tackle this problem. Bagging means that we create multiple models and average them. The twist is that for each model we draw a small subset of our data D (with replacement). Given this randomization in the training data, we can greatly lower the variance (Hastie et al. 2009a).

Consider M identically distributed (but not independent) trees with positive pairwise correlation ρ . We can write the variance of their average as (Hastie et al. 2009b):

$$\rho\sigma^2 + \frac{1-\rho}{M}\sigma^2 \quad (13)$$

As we increase the number of trees M , the right term of Equation 13 shrinks away. Unfortunately the correlation part $\rho\sigma^2$ still remains. Consequently, at some point we cannot significantly

reduce the variance any further by adding more trees into our ensemble.

In order to obtain even lower variance and thus better performance, one has to reduce the correlation between the trees. This effect can be achieved by only picking a random subset of of the p features in the data at each node split:

$$mtry \leq p$$

Every single tree is now forced to use different predictors for splitting at every node. Thereby, we now do not only construct bootstrap trees (i.e. trees with different training data), but also more de-correlated ones (Hastie et al. 2009b). Algorithm 2 sums up how the random forest works. Eventually we obtain an ensemble of trees. For regression, we average the result of all trees.

Algorithm 2 Random Forest

```

1: Input: Data  $D$ , number of trees  $M$ , number of variables to draw at each split  $mtry$ 
2: for  $m = 1$  to  $M$  do
3:   Draw a bootstrap sample  $D^{[m]}$  from  $D$ 
4:   Fit a tree  $t^{[m]}(x)$  with data  $D^{[m]}$ 
5:   For each split consider only  $mtry$  randomly selected features
6:   Grow tree
7: end for
8: return ensemble of trees:  $\{t_m\}^M$ 
  To predict a new point  $x$ :
  Regression:  $\hat{f}^M(x) = \frac{1}{M} \sum_{m=1}^M t_m(x)$ 
  Classification: let  $\hat{C}_m x$  be the class prediction of the  $m$ th tree.
                 Then:  $\hat{C}^M(x) = \text{majority vote } \{\hat{C}_m(x)\}^M$ 

```

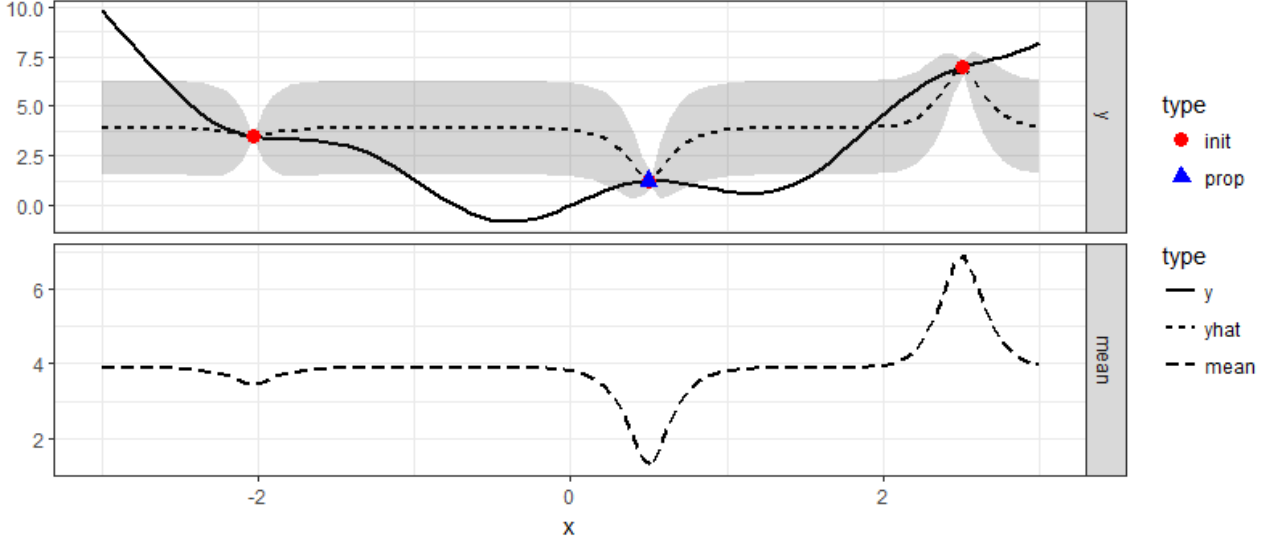
To utilize infill criteria, we need an estimation of the uncertainty. Like a Gaussian process, the random forest has this property naturally. In essence it is a consequence of the bagging procedure. We exploit this and compute the mismatch of each single tree’s prediction with the full ensemble:

$$\hat{\sigma}^2(x) = \frac{1}{M} \sum_{m=1}^M (\hat{t}^m(x) - \hat{f}^M(x))^2$$

In this equation $\hat{t}^m(x)$ corresponds to a tree which was trained with the m ’th bagging subset, whereas $\hat{f}^M(x)$ represents the full ensemble.

3.1.3 Infill Criteria

The infill criterion proposes new configurations. Perhaps the easiest variant is called “mean”. As the name suggests, it does only consider the mean which was estimated by the surrogate model. Figure 16a shows the problem of Equation 9. Our mean infill criterion proposes almost the same location where we already evaluated a configuration. We might even get stuck there.



(a)

Typically, the infill criterion does also include the uncertainty for its proposal. In other words, it controls the trade-off between exploration and exploitation. One of the most established variants is the expected improvement:

$$EI(x) = \mathbb{E}(I(x)) = \mathbb{E}(\max\{y_{min} - Y(x), 0\}) \quad (14)$$

We can think of this as the search for the greatest potential improvement over the currently best observed value y_{min} . $Y(x)$ is a random variable representing the posterior distribution at x . In the case of a Gaussian process surrogate model, the $Y(x)$ corresponds to a normal distribution. Figure 12a, b, c and d used the expected improvement for all of its proposals.

An easier alternative which we need in the last section of this chapter is called lower confidence bound:

$$LCB(x, \lambda) = \hat{\mu}(x) - \lambda \hat{\sigma}(x) \quad (15)$$

The trade-off between the exploitation and exploration is only controlled by a parameter λ . Note that the infill criterion itself requires optimization as well. The mlrMBO package (Bischl et al. 2017) uses a technique called focus search. It basically constructs a huge random design whereof all points are being evaluated by the surrogate model. Following up, the area next to the most promising configuration is shrunk down and the procedure repeated to enforce local convergence.

3.2 Hyperband

In Bayesian Optimization, we gradually propose new configurations.

Hyperband turns the tables as it instead gradually allocates resources across a set of randomly sampled configurations. For resources, we can think of a budget which is appropriate to the algorithm. In the world of boosting, these resources could be the total number of boosting iterations. A neural network on the other hand has two obvious alternatives: the epochs or the mini batch updates. Also, a temporal resource type would work for both methods.

3.2.1 Successive Halving and the B to n Problem

Li et al. (2016) establish the Hyperband algorithm as a solver for multi-armed bandit problems. In order to understand this, imagine a gambler in a casino who wants to maximize his profit. He is free to choose between n different slot machines and has a fixed budget B . At first glance, he has absolutely no knowledge of each slot machine’s rate of return. So he begins to spend b units of budget for each machine, such that $b \cdot n \ll B$. To put it another way, this means “pulling” each bandits arm b times. As a consequence, he loses some of his budget B . Likewise, he does also gain a bit of information concerning each machine’s return rates. The upcoming question for the gambler is how to allocate his remaining budget on the bandits.

To translate this idea into our hyperparameter optimization problem, we simply think of configurations instead of armed bandits. This means that a limited amount of resources must be distributed across competing configurations such that we optimize our performance measure. Initially, each configuration’s capabilities are only little known. As we begin to carefully allocate more resources, each configuration might become better understood.

One way to solve the question on how to allocate the resources could be provided by Successive Halving (Jamieson & Talwalkar 2015). Algorithm 3 shows the mechanics.

Summarized, all Successive Halving is doing is to evaluate each configuration and then to eliminate the worse half according to the performance measure. Subsequently, more budget will be allocated to the victors. The whole procedure is repeated for $\lceil \log_2 (n) \rceil$ times, where n represents the total number of initial arms.

Algorithm 3 Successive Halving

```
1: input: total budget  $B$ 
2: input:  $n$  arms
3: initialize:  $S_0 \leftarrow \{1, 2, 3, \dots, n\}$ 
4: for  $r = 0$  to  $(\lceil \log_2 (n) \rceil - 1)$  do
5:   sample each arm  $i \in S_r$  for
      
$$b_r = \left\lfloor \frac{B}{|S_r| \lceil \log_2 (n) \rceil} \right\rfloor$$

      times, and let  $\hat{p}_i^r$  be the reward
6:   let  $S_{r+1}$  be the set of  $\lceil \frac{1}{2} \cdot S_r \rceil$  arms with the largest reward
7: end for
8: return arm in  $S_{\lceil \log_2 (n) \rceil}$ 
```

This, and the budget allocation of line 5 of Algorithm 3, ensure that the entire budget will never be exceeded.

Suppose we had a total budget of $B = 1000$ and $n = 100$ arms. Table 3 shows the procedure of the Successive Halving algorithm. At first, we allocate $b_0 = 1$ unit of budget to each of our $n = 100$ arms. Subsequently, we eliminate the worst 50 arms from our investigation. To each of our remaining arms, we allocate 2 additional resources. Equation 16 shows how to calculate the budget allocation for the second round $r = 1$. $\lfloor \cdot \rfloor$ represents the floor and $\lceil \cdot \rceil$ the ceiling function respectively.

$$\begin{aligned}
b_r &= \left\lfloor \frac{B}{|S_r| \lceil \log_2(n) \rceil} \right\rfloor \\
&= \left\lfloor \frac{1000}{\underbrace{|S_1|}_{|\{1,2,\dots,50\}|} \lceil \log_2(100) \rceil} \right\rfloor = 2
\end{aligned} \tag{16}$$

After a total of $|r| = 7$ iterations, we obtain the best arm according to the Successive Halving criterion. Overall, we spent 877 of our 1000 resources.

Table 3: Example for Successive Halving. For a budget of $B = 1000$ and $n = 100$ arms, we conduct a total of $|r| = 7$ iterations. In the first one, each of our arms receives 1 unit of budget. Next, we filter the worst 50 arms and allocate another 2 units of budget to the victors.

iteration r	number of arms left	budget b_r for each arm	total budget spent
0	100	1	100
1	50	2	200
2	25	5	325
3	13	10	455
4	7	20	595
5	4	35	735
6	2	71	877

Assume we would like to find a good set of hyperparameters for a neural network.

To do this, we are limited by a fixed budget B . In order to conduct Successive Halving, we have to select a number of configurations n which we would like to try. As a consequence, we implicitly decide the amount of resources b_0 which are allocated to each configuration. Therefore, we also determine how “long” we examine each configuration, until the “bad” ones are being sorted out. For the previous example of Table 3, we obtained $b_0 = 1$.

Consider Figure 17. It shows the error-behavior of two arbitrary neural network configurations on the same data. Both curves intersect at roughly 10 training iterations (e.g. epochs). While “Model B” manages to decrease its validation error faster than “Model A”, it does also level off at a higher terminal error. This means in particular, according to the Successive Halving criterion, we would very likely opt for the inferior “Model B”.

To evade the wrong choice in this situation, we required $b_0 \geq 10$. For a budget of $B = 1000$, that only happens if we set $n \leq 20$ (see Equation 17).

$$b_0 = \left\lfloor \frac{1000}{|S_0| \lceil \log_2(n) \rceil} \right\rfloor \stackrel{!}{\geq} 10 \quad (17)$$

$$\Leftrightarrow n \leq 20$$

This is called the B to n problem. We do not know how to choose n appropriately in order to obtain the best possible configuration. Choosing n too high will very likely lead to wrong decisions. Selecting a value which is too low means that we hardly experience a sufficient range of configurations. Hyperband captures the basic idea of Successive Halving and refines it in order to tackle the B to n problem.

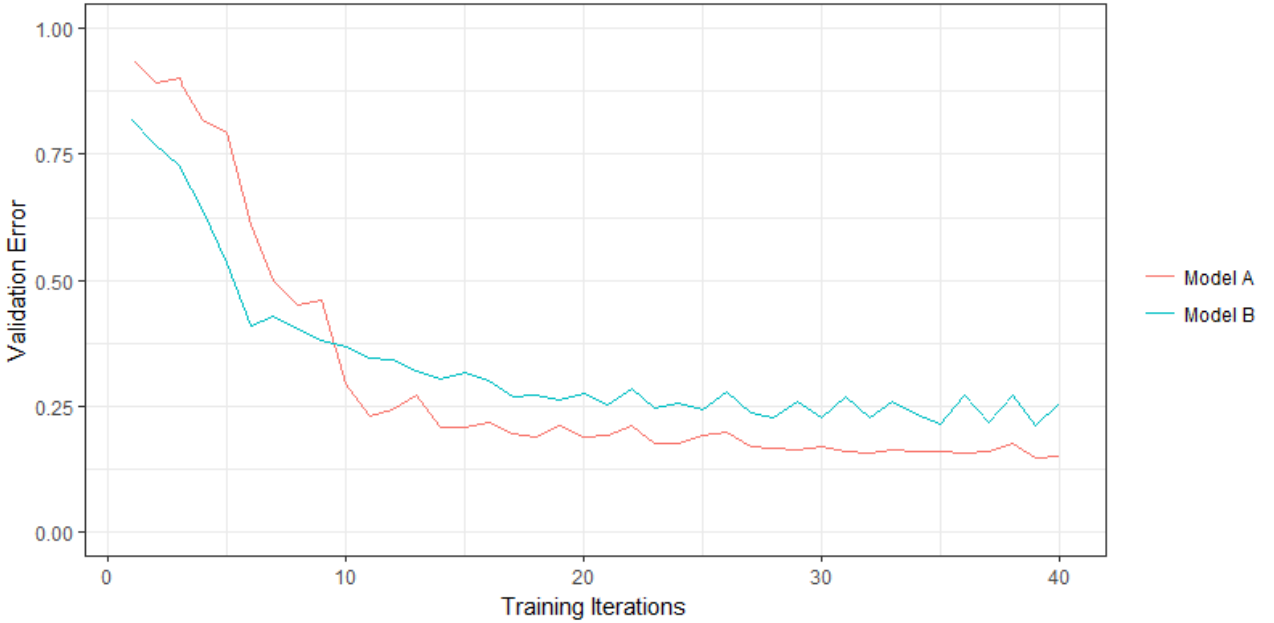


Figure 17: The B to n problem. In red and turquoise, we see the validation loss of two arbitrary neural networks on the same data. While “Model B” manages to learn faster, it also levels off at a higher error than “Model A”. Thus, if we conduct Successive Halving and set n to high, we opt for the wrong configuration.

3.2.2 The Hyperband Algorithm

To pursue the intent of overcoming the B to n problem, Hyperband introduces so-called brackets. Each bracket is a slight modification of the Successive Halving algorithm. The main difference is that all brackets consider a different value for n .

Algorithm 4 describes the procedure. Our new input parameters are the two integers R and η .

Algorithm 4 Hyperband

```
1: input:  $R = \text{max.resources}$ ,  $\eta = \text{prop.discard}$ 
2: initialize:  $s_{\max} = \lfloor \log_{\eta}(R) \rfloor$  and  $B = (s_{\max} + 1) \cdot R$ 
3: for  $s \in \{s_{\max}, s_{\max} - 1, \dots, 0\}$  do
4:   initialize:  $n = \lceil \frac{B}{R} \cdot \frac{\eta^s}{(s+1)} \rceil$  and  $r = R \cdot \eta^{-s}$ 
5:   sample:  $T = \text{get\_hyperparameter\_configurations}(n)$ 
   inner loop: begin Successive Halving with  $(n, r)$ 
6:   for  $i \in \{0, \dots, s\}$  do
7:      $n_i = \lfloor n \cdot \eta^{-i} \rfloor$ 
8:      $r_i = r \cdot \eta^i$ 
9:      $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$ 
10:     $T = \text{get\_top\_k\_models}(T, L, k = \lfloor \frac{n_i}{\eta} \rfloor)$ 
11:   end for
12: end for
13: return
```

R represents the maximum amount of resources which can be allocated to a single configuration. The second one, η , is a control parameter. It proportions the number of configurations to be discarded in each round of Successive Halving. In the proper meaning of the word, it is actually only “halving” when we set $\eta = 2$. Thus, Hyperband enables us to alter the fraction of configurations which we would like to sort out.

As we call Hyperband, two important values will be initialized (line 2 of algorithm 4). This is the total number of brackets $s_{\max} + 1$ as well as the amount of budget B each bracket is allowed to spend.

Let us work through the first bracket $s = s_{\max}$. This is the most “aggressive” one, as it sets n to maximize the exploration and thereby sorts out quite fast. The n configurations T will be sampled from a distribution which is defined over the configuration space. This could either be a simple hypercube with minimum and maximum boundaries for each hyperparameter, but also a distribution which includes some useful priors. In line 9, we begin to train each configuration $t \in T$ for r_i units of budget. Based on their validation losses, we update T such that it now only contains the best $k = \lfloor \frac{n_i}{\eta} \rfloor$ configurations. The second round $i = 1$ repeats the procedure and allocates more resources to the more promising configurations.

Summing up, we have an outer loop, iterating across several distinct brackets and an inner loop, conducting Successive Halving in each of these brackets.

3.2.3 Hyperband Example

Suppose we would like to call Hyperband with $R = 81$ and $\eta = 3$. According to the algorithm, we initialize

$$s_{\max} = \lfloor \log_{\eta}(R) \rfloor = \lfloor \log_3(81) \rfloor = 4$$

$$B = (s_{\max} + 1) \cdot R = (4 + 1) \cdot 81 = 405$$

Since the outer loop ranges from s_{\max} to 0, we obtain a total of $s_{\max} + 1 = 5$ brackets. Each of

these brackets are allowed to spend up to $B = 405$ resources.

Hyperband begins with the most aggressive bracket $s = s_{max} = 4$. The first step is to sample

$$\begin{aligned} n &= \left\lceil \frac{B}{R} \cdot \frac{\eta^s}{(s+1)} \right\rceil \\ &= \left\lceil \frac{405}{81} \cdot \frac{3^4}{(4+1)} \right\rceil = 81 \end{aligned}$$

configurations from the predefined hyperparameter space. In Addition to that, we state the initial budget for each configuration in this bracket:

$$\begin{aligned} r &= R \cdot \eta^{-s} \\ &= 81 \cdot 3^{-4} = 1 \end{aligned}$$

Now we start with the first round of Successive Halving, that is $i = 0$. At each iteration i , we assign the current number of configurations n_i :

$$\begin{aligned} n_i &= \lfloor n \cdot \eta^{-i} \rfloor \\ n_0 &= \lfloor 81 \cdot 3^{-0} \rfloor = 81 \end{aligned}$$

The same has to be done for r_i :

$$\begin{aligned} r_i &= r \cdot \eta^i \\ r_0 &= 1 \cdot 3^0 = 1 \end{aligned}$$

Trivially, for $i = 0$, n_i as well as r_i are equal to n and r respectively.

Up next, we evaluate each configuration and keep the best:

$$k = \left\lfloor \frac{n_i}{\eta} \right\rfloor = \left\lfloor \frac{81}{3} \right\rfloor = 27$$

To conduct the second round, we assign the new values for n_1 and r_1 :

$$\begin{aligned} n_1 &= \lfloor 81 \cdot 3^{-1} \rfloor = 27 \\ r_1 &= 1 \cdot 3^1 = 3 \end{aligned}$$

This time we keep the best

$$k = \left\lfloor \frac{27}{3} \right\rfloor = 9$$

configurations. For $i \in \{0, \dots, s\}$ and $s = s_{max} = 4$, we experience a total of 5 rounds of Successive Halving. Eventually, the surviving configuration is allocated a total of $R = 81$ resources. Figure

18 shows the entire results of all brackets. We see that bracket 1 completely exhausted the allowed amount of $B = 405$ units of budget. The second, third and fourth bracket do not entirely consume their budgets. In the fifth and last bracket, we sample only $n_i = 5$ configurations. Each of them are allocated $r_i = 81$ units of budget.

i	$s = 4$		$s = 3$		$s = 2$		$s = 1$		$s = 0$	
	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	34	3	15	9	8	27	5	81
1	27	3	11	9	5	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								
total B	405		363		351		378		405	

Figure 18: Hyperband with $R = 81$ and $\eta = 3$. The first bracket $s = 4$ samples $n_i = 81$ configurations. It discards the worst $\frac{2}{3}$ after each configuration has been allocated merely $r_i = 1$ units of budget. Bracket $s = 0$ on the other hand does almost exactly the opposite. While it only samples $n_i = 5$ configurations, all of them are trained for $r_i = 81$ units of budget before Successive Halving is executed.

3.3 Combining Hyperband with Model-Based Optimization

This chapter presents a possible way of combining Hyperband with Bayesian Optimization. In particular, we want to improve the selection of configurations.

Recall that the conventional Hyperband algorithm samples a different amount n of new configurations in each bracket. In essence, this tackles the B to n problem.

However, as we decrease n , the risk of having proportionally more bad than good configurations increases. Assume that we have large boundaries for a couple of hyperparameters in our search space (e.g. bad prior knowledge). This will amplify the effect and hence, the rear brackets will very likely waste many resources on a bad configuration.

Figure 19a shows this problem while optimizing a neural network with Hyperband. Each dot in this plot represents a configuration at a certain state and bracket (the budget allocation). These interconnected dots are the configurations which were not sorted out immediately but retrained instead. Both rear brackets $s = 1$ and $s = 0$ sample predominantly “bad” configurations. The accuracy of each bracket’s best configuration are 0.962, 0.953, 0.939, 0.881 and 0.89 (from $s = 4$ to $s = 0$). Hence, the most exploratory bracket $s = 4$ yielded the best performance.

In order to approach this problem, we propose to transfer information between each bracket. Figure 20 illustrates the basic idea. For Hyperband input parameters $R = 81$ and $\eta = 3$, we obtain a total of 5 brackets. The first one, $s = 4$, samples $n = 81$ configurations. Successive halving enables us to observe the behaviour for some of these configurations at different “budget stages”. Our central idea is to exploit this information and fit a surrogate model to learn

$$\underbrace{\mathbb{X}}_{\text{configurations}} \times \underbrace{\mathbb{R}}_{\text{budget}} \rightarrow \underbrace{\mathbb{R}}_{\text{performance}} \quad (18)$$

In particular, we want to use the most exploratory bracket as an initial design.

For the example of Figure 20, we know for at least $n = 27$ configurations how they performed with $r_0 = 1$ and $r_1 = 3$ units of budget. The best configurations of this bracket even contain 5 different values of r_i .

Suppose we were optimizing neural networks and had a search space with three hyperparameters. The evolution of the best configuration’s accuracy in bracket $s = 4$ might look like this:

##	optimizer	learning.rate	batch.normalization	current_budget	y
## 1	adam	0.05690947	TRUE	1	0.456
## 2	adam	0.05690947	TRUE	3	0.727
## 3	adam	0.05690947	TRUE	9	0.912
## 4	adam	0.05690947	TRUE	27	0.960
## 5	adam	0.05690947	TRUE	81	0.975

So for $r_0 = 1$, the architecture obtains an accuracy of 45.6%. But for $r_1 = 3$, the same configuration yields 72.7%. According to Equation 18, we utilize the budget as an additional hyperparameter. Furthermore, we take the complete bracket $s = 4$ to represent our initial design. As a result, this design contains a total of $n = 121$ configurations ($81 + 27 + 9 + 3 + 1 = 121$).

Based on these, we want to propose $n = 34$ configurations for the subsequent bracket $s = 3$. In particular, we want configurations which are good for the maximum amount of resources in this bracket. So in order to conduct multi-point proposals, we need a purpose-built infill criterion. Actually invented to parallelize and therefore accelerate Bayesian Optimization, qLCB (Hutter et al. 2012) provides exactly what we need. Recall LCB from Equation 15. A simple trade-off between exploration and exploitation is controlled by a parameter λ . Now instead of one fixed value for λ , qLCB samples k different values λ_k from an exponential distribution. With parameter $\frac{1}{\lambda}$ and expected value λ we would like to sample:

$$\begin{aligned} qLCB(x, \lambda_k) &= \hat{\mu}(x) - \lambda_k \hat{se}(x), \\ \lambda_k &\sim \text{Exp}\left(\frac{1}{\lambda}\right), k = 1, \dots, m \end{aligned} \tag{19}$$

Each of these values for λ_k is optimized separately. In essence, low values of λ_k mean exploitation while high values on the other hand mean exploration.

Figure 19b shows the effect of this strategy on the same problem of Figure 19a. We see that the new sampling strategy avoids bad configurations in the rear brackets. In this particular example, the best performance was even observed in bracket $s = 1$ (accuracy of 0.973).

We call this method “Hyperband + MBO Budget”.

Supplementary, we want to propose two additional variants to combine Hyperband with MBO. For both of them, their surrogate models try to learn the simplified space

$$\underbrace{\mathbb{X}}_{\text{configurations}} \rightarrow \underbrace{\mathbb{R}}_{\text{performance}} \tag{20}$$

The first of them which we call “Hyperband + MBO Mean” aggregates configurations based on their mean performance. Every configuration is weighted equally in the mean independent of the budget used for its evaluation. For instance, the configuration of the best model of bracket $s = 4$:

##	optimizer	learning.rate	batch.normalization	current_budget	y
## 1	adam	0.05690947	TRUE	1	0.456
## 2	adam	0.05690947	TRUE	3	0.727
## 3	adam	0.05690947	TRUE	9	0.912
## 4	adam	0.05690947	TRUE	27	0.960
## 5	adam	0.05690947	TRUE	81	0.975

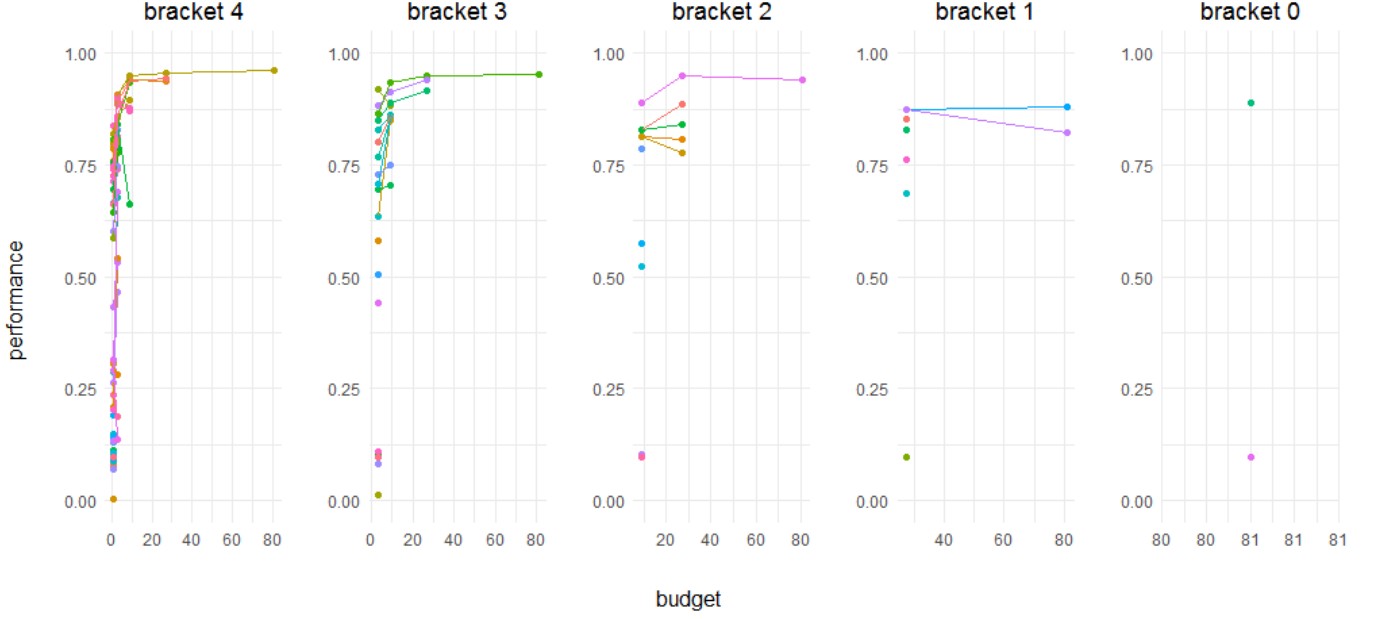
will be reduced to

##	optimizer	learning.rate	batch.normalization	y_mean.perf
## 1	adam	0.05690947	TRUE	0.806

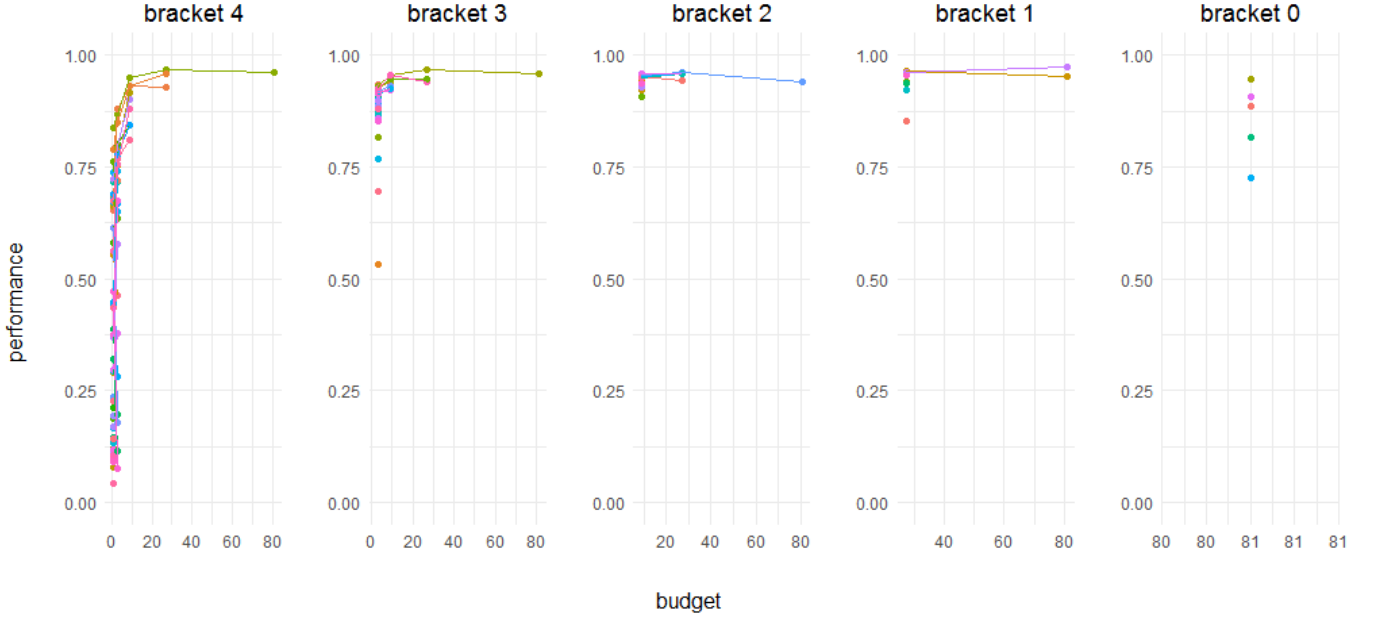
The second modification “Hyperband + MBO Max” aggregates multiple occurring configurations by only using the best observed performance instead:

```
## optimizer learning.rate batch.normalization y_max
## 1 adam 0.05690947 TRUE 0.975
```

We will evaluate all of these mutations in chapter 5 on several problems.



(a) Optimizing a neural network with Hyperband. Here we suppose bad prior knowledge for some of our hyperparameters. Thus we have to deal with larger boundaries in our search space. Bracket $s = 1$ and $s = 0$ issue a lot of resources on bad configurations. The best performance was achieved by bracket $s = 4$, the most exploratory one (accuracy of 0.962).



(b) Optimizing a neural network with the combination of MBO and Hyperband on the same problem as Figure 19a. We utilized the budget as an additional hyperparameter (see Equation 18 and Figure 20). All brackets do now exhibit very similar performance values. The best performance was achieved by bracket $s = 1$ (accuracy of 0.973)

Figure 19: Hyperband vs Hyperband in combination with MBO, given bad prior knowledge.

i	$s = 4$		$s = 3$		$s = 2$		$s = 1$		$s = 0$	
	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	34	3	15	9	8	27	5	81
1	27	3	11	9	5	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								




Figure 20: Information transfer between Hyperband brackets. We want to utilize the configuration's of bracket $s = 4$ as our initial design. Equation 18 states that we also include each configurations budget r_i as an additional hyperparameter for our surrogate model. Thus, we obtain an initial design with $81 + 27 + 9 + 3 + 1 = 121$ configurations. Following up, a multi-point infill criterion will propose $n = 34$ configurations for the upcoming bracket $s = 3$. After bracket $s = 3$ has finished, we add these $34 + 11 + 3 + 1 = 49$ configurations into our design. The design now contains $121 + 49 = 170$ configurations. Hence, the configurations for bracket $s = 2$ will be based on even more information than those of bracket $s = 3$. Bracket $s = 1$ on more than $s = 2$ and $s = 0$ on more than $s = 1$.

4 Implementation

One crucial part of this thesis was the implementation of the Hyperband algorithm. Since we planned further extensions of the original version, our central goal was to obtain a very generic variant. Therefore we opt for R6 (Chang 2017).

4.1 A Brief Introduction to R6

R6 is based on an encapsulated object oriented system, similar to those in Python or Java. In addition to data, objects do now contain methods. We can think of these methods as functions or “abilities” of the object. Using these methods enables us to directly modify the object. Suppose we would like to design an R6 class that creates objects for anytime algorithms. When initialized, the “algorithm objects” should contain a model as well as a method to continue the training of the model. The class could look like this:

```
algorithm = R6Class("Algorithm",  
  # The public argument is always a list and defines the interface of the object.  
  public = list(  
    my.data = NULL,  
    initial.budget = NULL,  
    current.budget = NULL,  
    init.fun = NULL,  
    train.fun = NULL,  
    model = NULL,  
    # To construct a new algorithm object, we have to call the $new() method.  
    # It will automatically invoke the $initialize() method. Thus, to create an  
    # algorithm object, we have to input four arguments:  
    initialize = function(my.data, initial.budget, init.fun, train.fun) {  
      self$my.data = my.data  
      self$current.budget = initial.budget  
      self$model = init.fun(self$my.data, self$current.budget)  
      self$train.fun = train.fun  
    },  
    # For an arbitrary budget, the $train method will call the train.fun:  
    train = function(budget) {  
      self$model = self$train.fun(self$model, self$my.data, budget)  
      self$current.budget = self$current.budget + budget  
    }  
  )  
)
```

By calling the `$new()` method, we can create algorithm objects. The main ingredients are the `init.fun`, a function to initialize models and a `train.fun` to access the train method:

```
myAlgorithmObject = algorithm$new(my.data = my.data,
                                  initial.budget = 10,
                                  init.fun = init.fun,
                                  train.fun = train.fun)
```

This is where the advantage of R6 manifests itself. We chose an `init.fun` such that we obtain a boosting ensemble from the XGBoost package (Chen et al. 2018):

```
myAlgorithmObject

## <Algorithm>
##   Public:
##     current.budget: 10
##     initialize: function (my.data, initial.budget, init.fun, train.fun)
##     model: xgb.Booster
##     my.data: xgb.DMatrix
##     train: function (budget)
```

But we could have chosen anything else, such as a neural network from the MXNet package (Chen et al. 2017) in combination with mlr (Bischl et al. 2016). All we have to change are our `init.fun` and `train.fun`:

```
myOtherAlgorithmObject

## <Algorithm>
##   Public:
##     current.budget: 10
##     initialize: function (my.data, initial.budget, init.fun, train.fun)
##     model: classif.mxff from package mxnet
##     my.data: list
##     train: function (budget)
```

By calling the `train` method, we directly manipulate the object. That means that we alter the `current.budget`, but in particular, the state of the model.

```
myAlgorithmObject$train(20)

## <Algorithm>
##   Public:
##     current.budget: 30
##     initialize: function (my.data, initial.budget, init.fun, train.fun)
##     model: xgb.Booster
##     my.data: xgb.DMatrix
##     train: function (budget)
```

In essence, we obtain an universal implementation, which is working with every other R package (as long it meets the requirements of the Hyperband algorithm).

4.2 The hyperbandr R Package

Figure 21 shows the general working flow of the hyperbandr package.

Calling `hyperband` will create brackets as R6 classes. Each bracket object itself will fabricate multiple algorithm objects. The hyperband pipeline will also construct different storage objects. For all configurations and in particular at all states (e.g. when we allocate more resources), they store and thus track the performance shift.

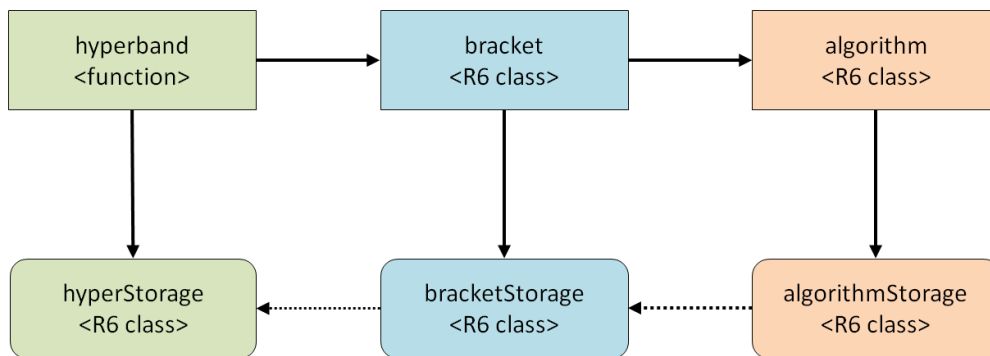


Figure 21: The working flow of the hyperbandr package. Calling `hyperband` will create bracket objects. Each bracket creates multiple algorithm objects. The package comes along with different storage objects. These store the configurations and track the changes in the appropriate performance at each state of the model (e.g. when we allocate more budget).

4.2.1 Algorithm Objects

We introduce the algorithm object as the fundamental building block of the package.

Algorithm objects are very similar to the R6 class which was displayed in section 4.1.

Figure 22 illustrates what inputs are required in order to create an algorithm object. We need an `init.fun` to initialize models and a `train.fun` to continue the training of a model.

An important enhancement is a function to evaluate the performance of the current model. This `perf.fun` can include any custom tailored objective function. Think of the MSE for regression or the logistic loss for a binary classification problem.

To realize Hyperband, we have to incorporate different configurations for our algorithms. As a consequence, we add the configuration to our list of inputs. Last but not least we would also like to input a unique id, as well as a variable called `problem`. The latter one includes the data. If required, we may also add the resampling instructions.

Each algorithm object has three methods. After each round of successive halving, the original Hyperband algorithm trains models from scratch (see section 3.2). However, in order to save computational time, we decided to retrain our models. Because of that, we add the `$continue(budget)` method to continue the training of a model for an deterministic amount of budget. In addition to that, the `$getPerformance()` method computes the performance measure

of the model at its current state. Moreover, there is a method called `$visPerformance()` to immediately plot the performance of the model.

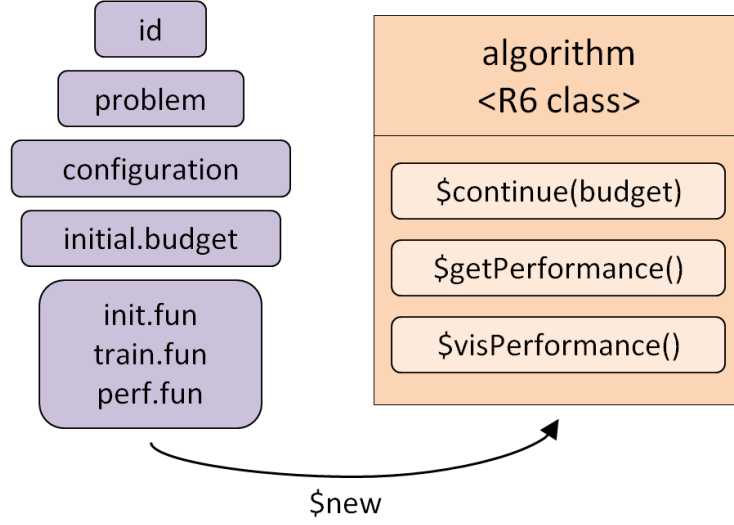


Figure 22: The algorithm object requires seven inputs. That is a unique **id**, a **problem**, containing the data and if necessary the resampling rule, as well as the **configuration** of the algorithm. In addition to that, we need an **initial budget** and three functions. One to **initialize a model**, one to **continue the training** and one to measure the **performance** of the model. Each algorithm object is equipped with three methods. Calling `$continue(budget)` will continue the training for `<budget>` resources. The other methods aim to compute and visualize the performance.

4.2.2 Bracket Objects

The heart of the package is the bracket class. Figure 23 describes its mode of operation in brief. Since the initial task of a bracket is to create a deterministic number of algorithm objects, we have to feed it almost the same arguments as we had to in chapter 4.2.1.

One major twist is that we obviously do not want to input a designated configuration. Instead, we need a search space. This is the `par.set` and it must include all hyperparameters as well as their feasible regions. Accordingly, we require a function to sample configurations from that search space. This `sample.fun` represents line 5 of Algorithm 4. We are free to deviate from a standard random sampling procedure and integrate other strategies in that function.

On top of that, each bracket object needs some additional Hyperband related parameters. These comprise the bracket index s , which affects the total number of configurations to be sampled. But also the B , which represents the total amount of budget that can be spent in the appropriate bracket.

The different bracket mechanics are obtained by a variety of methods.

The `$step` method calls certain “submethods” to conduct one round of successive halving. At first, we eliminate the worst $1 - \frac{1}{k}$ configurations from our bracket object. This method may distinguish whether we would like to minimize (e.g. the MSE) or maximize (e.g. the accuracy) our performance measure. Subsequently, we feed more budget into each of the remaining $\frac{1}{k}$ configurations and call the `$continue` method.

Hyperband in hyperbandr actually just calls the `$run` method. It will perform successive halving until only one algorithm object, therefore the best configuration, is left.

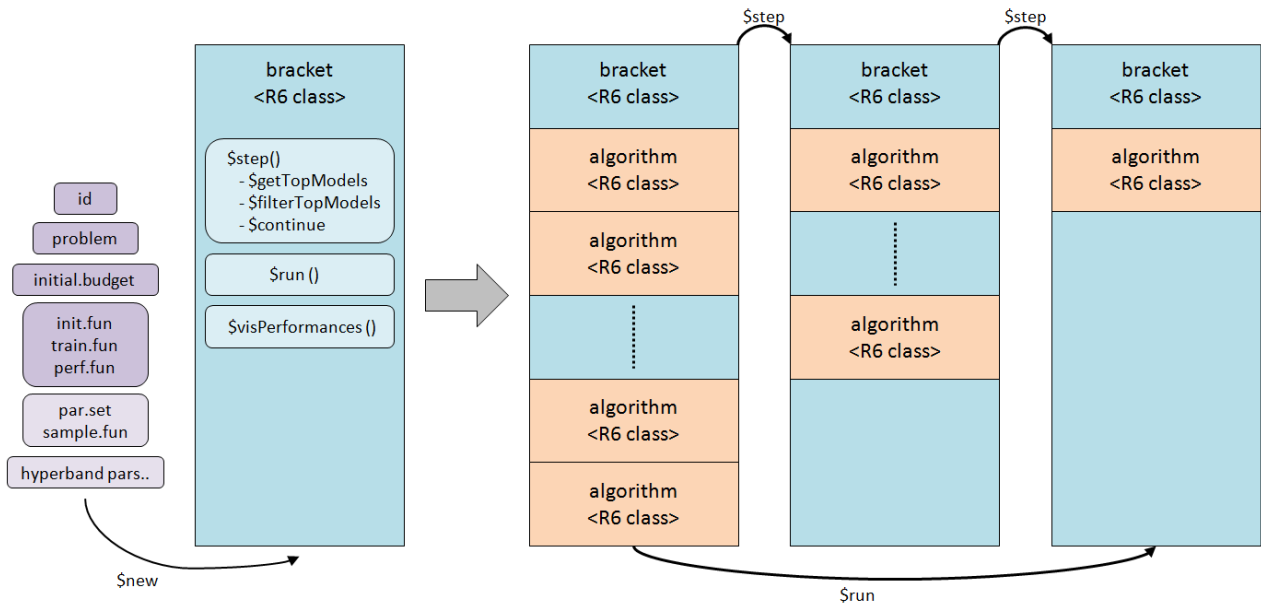


Figure 23: The working flow of a bracket object. In addition to the inputs we learned about in section 4.2.1, we require a search space as well as a function to sample from it. Each bracket begins with a deterministic number of algorithm objects. The `$run` method calls the `$step` method, which corresponds to one round of successive halving. This procedure is repeated until only one algorithm object, therefore the best configuration, is left.

4.2.3 Storage Objects

In order to provide more flexibility when it comes to the selection of new configurations, we include different storage objects (see Figure 21). These storage objects contain the hyperparameter configurations as well as their corresponding performance value. One may access these storage objects to propose new configurations by applying Bayesian Optimization strategies.

For instance, every algorithm object contains its own `algorithmStorage` object, which is another R6 class. Whenever we continue to train a particular configuration, we write a new line to its `algorithmStorage`. This way, we can nicely observe how the configuration behaves as we allocate more budget.

For an arbitrary neural network configuration, the `algorithmStorage` may look like this:

```
##      optimizer learning.rate batch.normalization current_budget  y
## 1      adam      0.05690947                TRUE              1 0.15
## 2      adam      0.05690947                TRUE              2 0.37
## ..
## 19     adam      0.05690947                TRUE             19 0.87
## 20     adam      0.05690947                TRUE             20 0.88
```

In this toy example, `current_budget` corresponds to the total epochs that we trained our network and the performance y to its current accuracy. What we see is as we gradually increase the budget by 1, the performance y does also improve.

Each bracket has a `bracketStorage` object. The storage binds all of its algorithm objects `algorithmStorages` and increases in size as we conduct successive halving. This happens simply because successive halving means allocating more resources to more promising configurations.

Consequently, we write lines to some `algorithmStorages` and eventually to the `bracketStorage`. When initializing a bracket object, its storage container may look like this:

```
##      optimizer learning.rate batch.normalization current_budget    y
## 1         sgd    0.02106619                TRUE          1 0.2
## 2         sgd    0.06116664                FALSE          1 0.0
## 3         adam    0.03720761                FALSE          1 0.3
## 4         sgd    0.06362754                TRUE          1 0.0
## 5         sgd    0.01780418                FALSE          1 0.1
## 6         sgd    0.06893814                TRUE          1 0.2
## 7         adam    0.03084838                FALSE          1 0.3
## 8         adam    0.02901885                FALSE          1 0.0
## 9         sgd    0.06961878                FALSE          1 0.1
## 10        sgd    0.05793493                FALSE          1 0.1
```

Moreover, as we call the `hyperband` function, an internally operating R6 class called `hyper.storage` will be created. This object concatenates bracket storage objects row-wise and is the easiest option to tap when planning to go for different sampling strategies.

4.3 The hyperbandr Vignette

The following subchapters are content of the `hyperbandr` package vignette. We begin to describe how to use the package in a general fashion. Following up, we present various examples of application.

4.3.1 Introductory Guide

In order to call `hyperband`, we have to customize a search space and four functions. Depending on the packages and problem at hand, particularly the latter ones may vary.

I: The Hyperparameter Search Space

At first, we design a hyperparameter search space. That search space includes all hyperparameters we would like to consider, as well as a reasonable range of values for each of them.

```
mySearchSpace = ...
```

II: The Sampling Function

Following up, we need a function to sample an arbitrary amount of hyperparameter configurations from our search space. The inputs of that function are:

- **par.set**: the search space
- **n.configs**: the number of configurations to sample
- **...**: additional arguments to access the hyper storage (see section 4.3.4 how to utilize this feature to combine Hyperband with Bayesian Optimization.)

```
sample.fun = function(par.set, n.configs, ...) {
  ...
}
```

The sampling function must return a list of named lists, containing the sampled hyperparameter configurations. For instance, the structure of the return value of our sampling function for an arbitrary example should look like this:

```
## List of 2
## $ :List of 3
## ..$ optimizer      : chr "adam"
## ..$ learning.rate   : num 0.0938
## ..$ batch.normalization: logi TRUE
## $ :List of 3
## ..$ optimizer      : chr "sgd"
## ..$ learning.rate   : num 0.0809
## ..$ batch.normalization: logi FALSE
```

III: The Initialization Function

We do also need a function to initialize our models. The inputs of that function must include:

- **r**: the amount of budget to initialize the model with
- **config**: a hyperparameter configuration
- **problem**: an object containing the data and if necessary a resampling rule

```
init.fun = function(r, config, problem) {
  ...
}
```

IV: The Training Function

The training function takes an initialized model and continues the training process. As described in chapter 4.2.1, instead of training a new model from scratch, we choose to continue training our existing model. That will greatly speed up training time. Necessary inputs are:

- **mod**: a model
- **budget**: the additional budget allocation
- **problem**: an arbitrary object containing the data and if necessary a resampling rule

```
train.fun = function(mod, budget, problem) {
  ...
}
```


V: The Performance Function

Our final ingredient is the performance function. That function simply evaluates the performance of the model at its current state. Its inputs are:

- **model**: a model to evaluate
- **problem**: an arbitrary object containing the data and if necessary a resampling rule

```
performance.fun = function(model, problem) {  
  ...  
}
```

Now that we have defined these functions, we can finally call hyperband. The inputs of hyperband are:

- **problem**: an arbitrary object containing the data and if necessary a resampling rule
- **max.resources**: the maximum amount of resources that can be allocated to a single configuration (the default is 81, that means in particular that we sample 81 configurations in our first bracket)
- **prop.discard**: a control parameter to define the proportion of configurations that will be discarded in each round of successive halving (the default is 3, that means in particular that we eliminate 2/3 in each round of successive halving)
- **max.perf**: a logical indicating whether we want to maximize (e.g. accuracy) or minimize (e.g. MSE) the performance measure
- **id**: a string generating an unique id for each model
- **par.set**: the hyperparameter search space
- **sample.fun**: the sampling function
- **init.fun**: the initialization function
- **train.fun**: the training function
- **performance.fun**: the performance function

```
myHyperbandr = hyperband(  
  problem = myProblem,  
  max.resources = 81,  
  prop.discard = 3,  
  max.perf = TRUE or FALSE,  
  id = "my id",  
  par.set = mySearchSpace,  
  sample.fun = sample.fun,  
  init.fun = init.fun,  
  train.fun = train.fun,  
  performance.fun = performance.fun  
)
```

4.3.2 Optimization of a Convolutional Neural Network with Hyperband

We would like to use a small subset of the original MNIST data (LeCun et al. 2010) and tune a neural network with hyperbandr. To this, we apply mlr (Bischl et al. 2016) as a wrapper for MXNet (Chen et al. 2017).

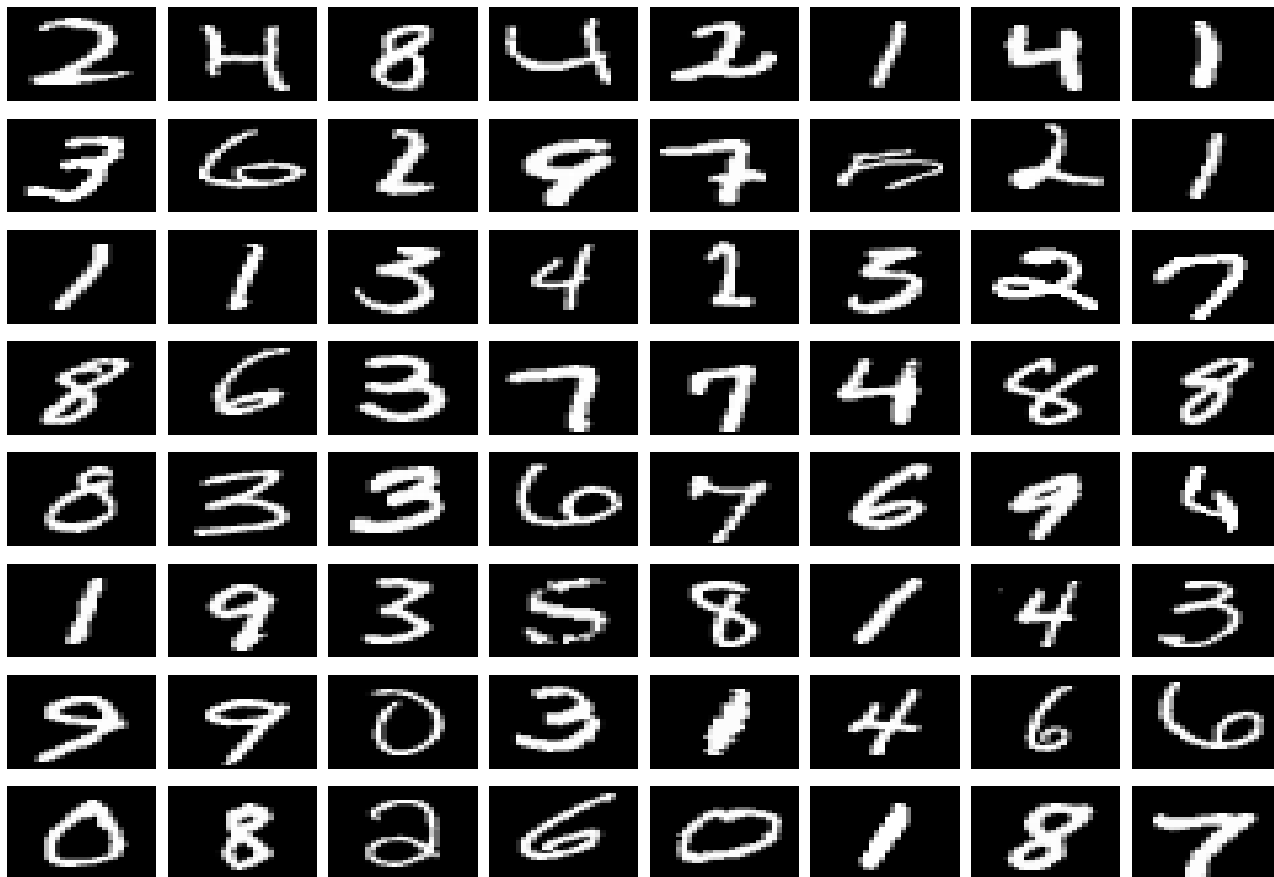


Figure 24: Snippet from the MNIST data that shows 64 different images (LeCun et al. 2010). The task is to classify handwritten digits on images. Thus, we have 10 different classes. For demonstration purposes, we only use 1/10 of the original data, which contains 60.000 images.

Our data has 6000 observations, evenly distributed on 10 classes:

```
dim(mnist)
```

```
## [1] 6000 785
```

```
table(mnist$label)
```

```
##  0  1  2  3  4  5  6  7  8  9
## 600 600 600 600 600 600 600 600 600 600
```

Let us create a list, which we call `problem`. That list should contain the data, as well as a resampling rule:

```

# We sample 2/3 of our data for training:
train.set = sample(nrow(mnist), size = (2/3)*nrow(mnist))

# Another 1/6 will be used for validation during training:
val.set = sample(setdiff(1:nrow(mnist), train.set), 1000)

# The remaining 1/6 will be stored for testing:
test.set = setdiff(1:nrow(mnist), c(train.set, val.set))

# Since we use mlr, we define a classification task to encapsulate the data:
task = makeClassifTask(data = mnist, target = "label")

# Finally, we define the problem list:
problem = list(data = task, train = train.set, val = val.set, test = test.set)

```

I. The Hyperparameter Search Space

The ParamHelpers package (Bischl et al. 2017) provides an easy way to construct the configuration space.

We opt to mainly search for regularization parameters, but also the optimizer as well as the learning rate (for all optimizers we choose the default values of MXNet). Boundaries for numeric parameters are given by the `lower` and `upper` argument.

```

library("ParamHelpers")
# We choose to search for optimal setting of the following hyperparameters:
configSpace = makeParamSet(
  makeDiscreteParam(id = "optimizer",
    values = c("sgd", "rmsprop", "adam", "adagrad")),
  makeNumericParam(id = "learning.rate", lower = 0.001, upper = 0.1),
  makeNumericParam(id = "wd", lower = 0, upper = 0.01),
  makeNumericParam(id = "dropout.input", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer1", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer2", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer3", lower = 0, upper = 0.6),
  makeLogicalParam(id = "batch.normalization1"),
  makeLogicalParam(id = "batch.normalization2"),
  makeLogicalParam(id = "batch.normalization3")
)

```

II: The Sampling Function

Now we need a function to sample configurations from our search space. We utilize the `sampleValues` function from the ParamHelpers package:

```
sample.fun = function(par.set, n.configs, ...) {
  # Sample multiple configurations from the par.set and remove all NAs.
  lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])
}
```

III: The Initialization Function

This function initializes a convolutional neural network with two convolutional as well as two dense layers. Note that we define layers = 3. The second dense layer is our output layer and will be automatically created by mlr.

One crucial aspect is the choice of our resource type, which we decide to be epochs (e.g. 1 unit of resources is one forward pass and one backward pass across the whole training set). In other words, when we initialize a model, we allocate **r** resources or train the network for **r** epochs.

```
init.fun = function(r, config, problem) {
  # We begin and create a learner.
  lrn = makeLearner("classif.mxff",
    # You have to install the gpu version of mxnet in order to run this code.
    ctx = mx.gpu(),
    layers = 3,
    conv.layer1 = TRUE, conv.layer2 = TRUE,
    conv.data.shape = c(28, 28),
    num.layer1 = 8, num.layer2 = 16, num.layer3 = 64,
    conv.kernel1 = c(3,3), conv.stride1 = c(1,1),
    pool.kernel1 = c(2,2), pool.stride1 = c(2,2),
    conv.kernel2 = c(3,3), conv.stride2 = c(1,1),
    pool.kernel2 = c(2,2), pool.stride2 = c(2,2),
    array.batch.size = 128,
    begin.round = 1, num.round = r,
    # This line is very important: here we allocate the configuration to our model.
    par.vals = config
  )
  # This will start the actual training (initialization) of the model.
  mod = train(learner = lrn, task = problem$data, subset = problem$train)
  return(mod)
}
```

IV: The Training Function

That function will take the initialized model and continues the training process.

To do this, most importantly, we have to extract the weights from our current model and assign them to a new learner.

This might seem needlessly complicated but will save us a lot of computational time. Other frameworks may include easier ways to retrain a pre-trained model, such that we can simply copy the init.fun.

```

train.fun = function(mod, budget, problem) {
  # We create a new learner and assign all hyperparameters from our current model.
  lrn = makeLearner("classif.mxff", ctx = mx.gpu(), par.vals = mod$learner$par.vals)
  lrn = setHyperPars(lrn,
    # In addition, we have to extract the weights and feed them into our new model .
    symbol = mod$learner.model$symbol,
    arg.params = mod$learner.model$arg.params,
    aux.params = mod$learner.model$aux.params,
    begin.round = mod$learner$par.vals$begin.round + mod$learner$par.vals$num.round,
    num.round = budget
  )
  mod = train(learner = lrn, task = problem$data, subset = problem$train)
  return(mod)
}

```

V: The Performance Function

The performance function will simply predict the validation data at each step of successive halving. Since we have evenly distributed class labels, we can optimize the accuracy without worries.

```

performance.fun = function(model, problem) {
  pred = predict(model, task = problem$data, subset = problem$val)
  # We chose accuracy as our performance measure.
  performance(pred, measures = acc)
}

```

Call hyperband

Now we can call hyperband. We take the default values for `max.resources` and `prop.discard`. Since we would like to maximize the accuracy, we obviously set `max.perf` to `TRUE`. Note: I do not recommend running this code on a CPU as even on a GTX 1070, it already takes around 7 minutes. If you do not own a GPU, substitute the convolutional layers with dense layers.

```

hyperbandr = hyperband(
  problem = problem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = TRUE,
  id = "myCNN",
  par.set = configSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)

```

```

## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Iteration 0, with 1 Algorithms left (Budget: 81)

```

Our output is a list of five brackets:

```
length(hyperbandr)
```

```
## [1] 5
```

We can inspect the first bracket ..

```
hyperbandr[[1]]
```

```

## <Bracket>
##   Public:
##     adjust: 27
##     B: 405
##     bracket.storage: BracketStorage, R6
##     clone: function (deep = FALSE)
##     configurations: list
##     filterTopKModels: function (k)
##     getBudgetAllocation: function ()
##     getNumberOfModelsToSelect: function ()
##     getPerformances: function ()

```

```
##      getTopKModels: function (k)
##      id: myCNN
##      initialize: function (problem, max.perf, max.resources, prop.discard, s,
##      iteration: 4
##      max.perf: TRUE
##      max.resources: NULL
##      models: list
##      n.configs: 1
##      par.set: ParamSet
##      printState: function ()
##      prop.discard: 3
##      r.config: 1
##      run: function ()
##      s: 4
##      sample.fun: NULL
##      step: function ()
##      visPerformances: function (make.labs = TRUE, ...)
```

.. and for instance check it's performance by calling the `$getPerformance()` method:

```
hyperbandr[[1]]$getPerformances()
```

```
## [1] 0.949
```

We can also inspect the architecture of the best model of bracket 1:

```
hyperbandr[[1]]$models[[1]]$model
```

```
## Model for learner.id=classif.mxff; learner.class=classif.mxff
## Trained on: task.id = mnist; obs = 4000; features = 784
## Hyperparameters: learning.rate=0.00674, array.layout=rowmajor,
  verbose=FALSE, optimizer=adam, wd=0.00386, dropout.input=0.188,
  dropout.layer1=0.172, dropout.layer2=0.147, dropout.layer3=0.38,
  batch.normalization1=FALSE, batch.normalization2=TRUE,
  batch.normalization3=TRUE, ctx=<MXContext>, layers=3,
  conv.layer1=TRUE, conv.layer2=TRUE, conv.data.shape=28,28,
  num.layer1=8, num.layer2=16, num.layer3=64, conv.kernel1=3,3,
  conv.stride1=1,1, pool.kernel1=2,2, pool.stride1=2,2,
  conv.kernel2=3,3, conv.stride2=1,1, pool.kernel2=2,2,
  pool.stride2=2,2, array.batch.size=128, begin.round=28,
  num.round=54, symbol=<Rcpp_MXSymbol>,arg.params=<list>,aux.params=<list>
```

Now let's check which bracket yielded the best performance:

```
lapply(hyperbandr, function(x) x$getPerformances())
```

```
## [[1]]  
## [1] 0.949  
##  
## [[2]]  
## [1] 0.948  
##  
## [[3]]  
## [1] 0.96  
##  
## [[4]]  
## [1] 0.975  
##  
## [[5]]  
## [1] 0.973
```

The hyperbandr package has a visualization function:

```
hyperVis(hyperbandr, perfLimits = c(0, 1))
```

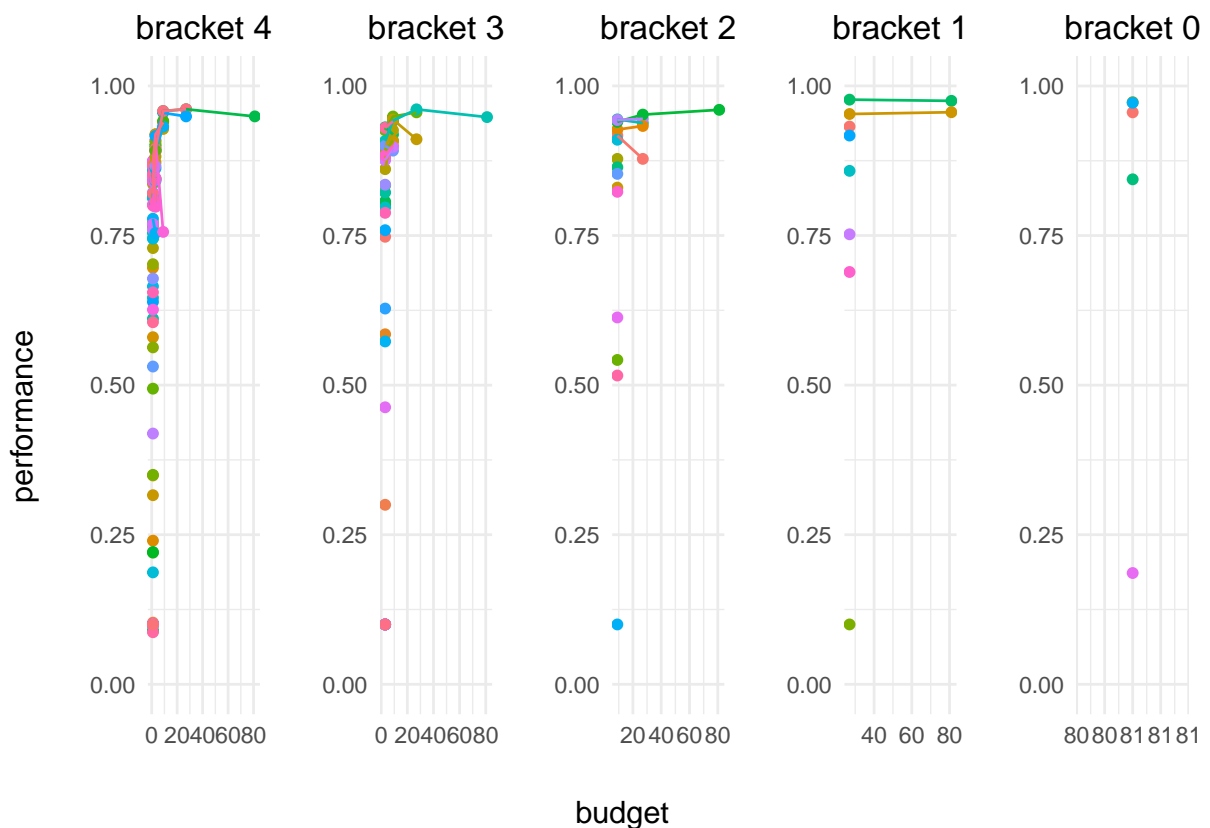


Figure 25: The hyperVis function visualizes each bracket in order to compare them. On the x-axis we see the budget and on the y-axis the performance. Each dot represents one configuration at a certain state. These interconnected dots are the configurations which were not sorted out immediately but retrained instead.

Finally, we use the best model over all brackets and predict the test data:

```
# Extract the best model:
best.mod.ind = which.max(unlist(lapply(hyperbandr, function(x) x$getPerformances()))))
best.mod = hyperbandr[[best.mod.ind]]$models[[1]]$model

# Predict the test data:
performance(predict(object = best.mod, task = problem$data, subset = problem$test),
             measures = acc)

## acc
## 0.978
```

4.3.3 Additional Features

The hyperbandr package can also compute single bracket objects. For demonstration purposes we shrink our hyperparameter search space. Computing a single bracket object requires us to input some new parameters:

- **s**: the s 'th bracket which we would like to compute (this will influence the number of configurations to be drawn)
- **B**: the (approximate) total amount of resources, which will be spend in that bracket

```
# Smaller config space for demonstration purposes.
configSpace = makeParamSet(
  makeDiscreteParam(id = "optimizer", values = c("sgd", "adam")),
  makeNumericParam(id = "learning.rate", lower = 0.001, upper = 0.1),
  makeLogicalParam(id = "batch.normalization"))

brack = bracket$new(
  problem = problem,
  max.perf = TRUE,
  max.resources = 81,
  prop.discard = 3,
  s = 4,
  B = (4 + 1)*81, # B = (sMax + 1) * max.resources
  id = "myBracket",
  par.set = configSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

Each bracket object has a bracket storage object which is basically just another R6 class. The bracket storage shows us the hyperparameters, the current budget and the performance in an equation-ish style.

```
# For max.resources = 81 and prop.discard = 3, we obtain 81 algorithms:
```

```
dim(brack$bracket.storage$data.matrix)
```

```
## [1] 81 5
```

```
# Print the first 10 configurations:
```

```
head(brack$bracket.storage$data.matrix, n = 10)
```

```
##      optimizer learning.rate batch.normalization current_budget      y
## 1         adam  0.056124215             FALSE             1 0.100
## 2          sgd  0.065107343              TRUE             1 0.184
## 3          sgd  0.083932671             FALSE             1 0.100
## 4          sgd  0.088801177              TRUE             1 0.672
## 5          sgd  0.008158432              TRUE             1 0.100
## 6          sgd  0.031451005              TRUE             1 0.100
## 7          sgd  0.033258030              TRUE             1 0.100
## 8         adam  0.032635399             FALSE             1 0.864
## 9         adam  0.009818755             FALSE             1 0.834
## 10        adam  0.091093379             FALSE             1 0.188
```

We could call the `$step()` method to conduct one round of successive halving. Or just complete the bracket by calling the `$run()` method. That means, that we conduct successive halving according to the rules described in section 4.2.

```
brack$run()
```

```
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
```

As we call the `$run()` method, we continuously write new lines to our bracket storage object.

```
# Hence, our bracket storage data matrix has 81 + 27 + 9 + 3 + 1 = 121 rows
```

```
dim(brack$bracket.storage$data.matrix)
```

```
## [1] 121 5
```

Our setup of `max.resources` and `prop.discard` means that our bracket storage contains 27 configurations twice, 9 three times, 3 four times and one even five times. They discern in its performance and `current_budget`.

```
# Print the last 10 configurations:
tail(brack$bracket.storage$data.matrix, n = 10)
```

##	optimizer	learning.rate	batch.normalization	current_budget	y
## 112	sgd	0.077285709	TRUE	9	0.962
## 113	adam	0.020103847	FALSE	9	0.934
## 114	adam	0.024215530	FALSE	9	0.936
## 115	adam	0.038770443	TRUE	9	0.937
## 116	adam	0.009818755	FALSE	9	0.949
## 117	adam	0.007867419	FALSE	9	0.944
## 118	sgd	0.077285709	TRUE	27	0.962
## 119	adam	0.041752407	TRUE	27	0.937
## 120	adam	0.009818755	FALSE	27	0.962
## 121	sgd	0.077285709	TRUE	81	0.972

Bracket objects have a `visPerformances()` method to immediately visualize the bracket.

```
brack$visPerformances()
```

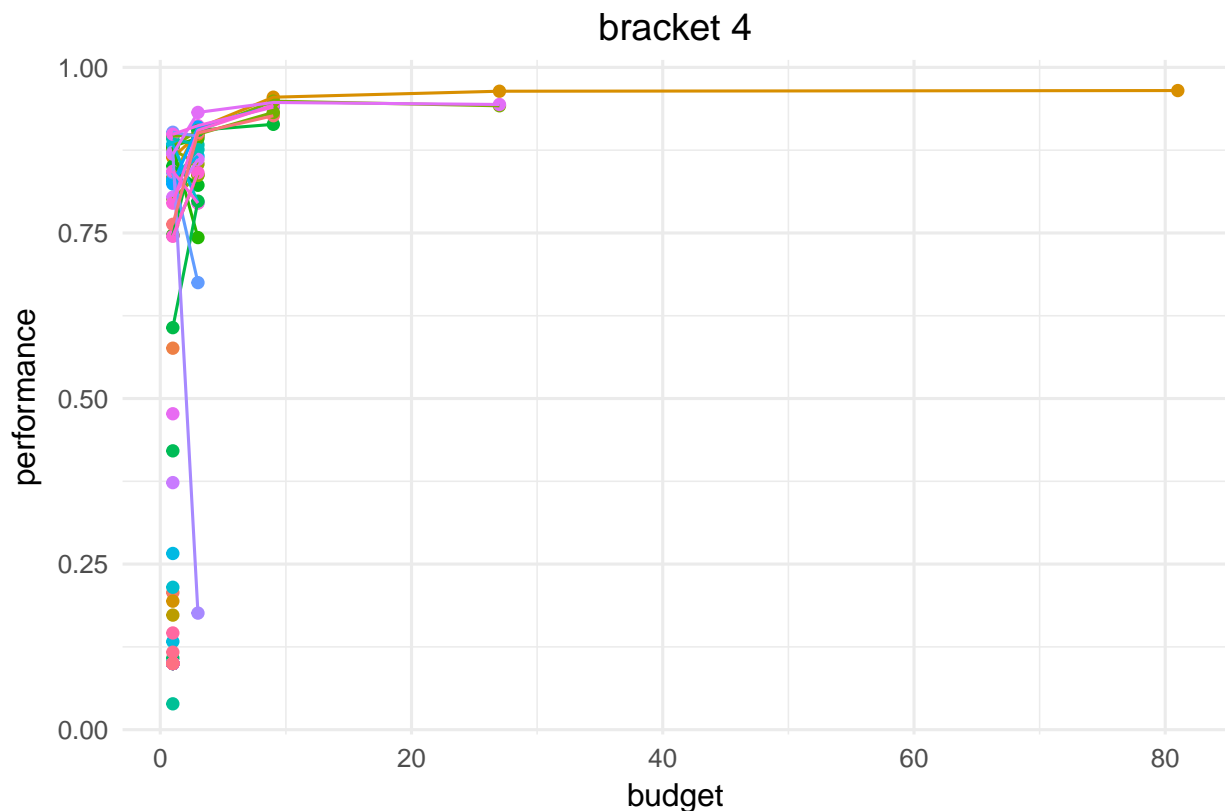


Figure 26: The `$visPerformance()` method visualizes the bracket. On the x-axis we see the budget and on the y-axis the performance. Each dot represents one configuration at a certain state. These interconnected dots are the configurations which were not sorted out immediately but retrained instead.

Beside the graphic investigation, we can also extract the best models performance by simply calling the `getPerformance()` method.

```
brack$getPerformances()
```

```
## [1] 0.965
```

Each bracket object contains multiple algorithm objects. The `hyperbandr` package allows us to create these algorithm objects solely and manipulate them. The input values are almost identical to those seen in the bracket object or when calling `hyperband`.

```
myConfig = sample.fun(par.set = configSpace, n.configs = 1)[[1]]
```

```
obj = algorithm$new(  
  problem = problem,  
  id = "myAlgorithmObject",  
  configuration = myConfig,  
  initial.budget = 1,  
  init.fun = init.fun,  
  train.fun = train.fun,  
  performance.fun = performance.fun)
```

We can inspect the configuration of our algorithm object:

```
# You can also call obj$model for much more details, but that would not fit on the page.  
obj$configuration
```

```
## $optimizer  
## [1] "adam"  
##  
## $learning.rate  
## [1] 0.05690947  
##  
## $batch.normalization  
## [1] TRUE
```

Similar to the bracket object, each algorithm object has an algorithm storage object which is basically just another R6 class. The algorithm storage shows us the hyperparameters, the current budget and the performance in an equation-ish style.

```
obj$algorithm.result$data.matrix
```

```
##   optimizer learning.rate batch.normalization current_budget    y  
## 1      adam   0.05690947                TRUE              1 0.856
```

The algorithm object does also have a `getPerformance()` method.

```
obj$getPerformance()
```

```
##    acc  
## 0.856
```

By calling the `continue()` method, we can continue training our algorithm object by an arbitrary amount of budget:

```
obj$continue(1)
```

Like before, in each step we write new lines to our algorithm storage. This enables us to track the behaviour of our algorithm object when allocating more resources.

```
obj$algorithm$result$data.matrix
```

```
##    optimizer learning.rate batch.normalization current_budget    y  
## 1      adam    0.05690947                TRUE              1 0.856  
## 2      adam    0.05690947                TRUE              2 0.927
```

So let us call `continue(1)` for 18 times to obtain a total of 20 epochs. This will write 18 additional lines to our algorithm storage:

```
##    optimizer learning.rate batch.normalization current_budget    y  
## 1      adam    0.05690947                TRUE              1 0.856  
## 2      adam    0.05690947                TRUE              2 0.927  
## 3      adam    0.05690947                TRUE              3 0.959  
## 4      adam    0.05690947                TRUE              4 0.960  
## 5      adam    0.05690947                TRUE              5 0.959  
## 6      adam    0.05690947                TRUE              6 0.965  
## 7      adam    0.05690947                TRUE              7 0.964  
## 8      adam    0.05690947                TRUE              8 0.964  
## 9      adam    0.05690947                TRUE              9 0.957  
## 10     adam    0.05690947                TRUE             10 0.955  
## 11     adam    0.05690947                TRUE             11 0.950  
## 12     adam    0.05690947                TRUE             12 0.936  
## 13     adam    0.05690947                TRUE             13 0.964  
## 14     adam    0.05690947                TRUE             14 0.963  
## 15     adam    0.05690947                TRUE             15 0.971  
## 16     adam    0.05690947                TRUE             16 0.960  
## 17     adam    0.05690947                TRUE             17 0.970  
## 18     adam    0.05690947                TRUE             18 0.962  
## 19     adam    0.05690947                TRUE             19 0.964  
## 20     adam    0.05690947                TRUE             20 0.975
```

To visualize the training process and the development of our validation error, we simply call the `visPerformance()` method:

```
obj$visPerformance()
```

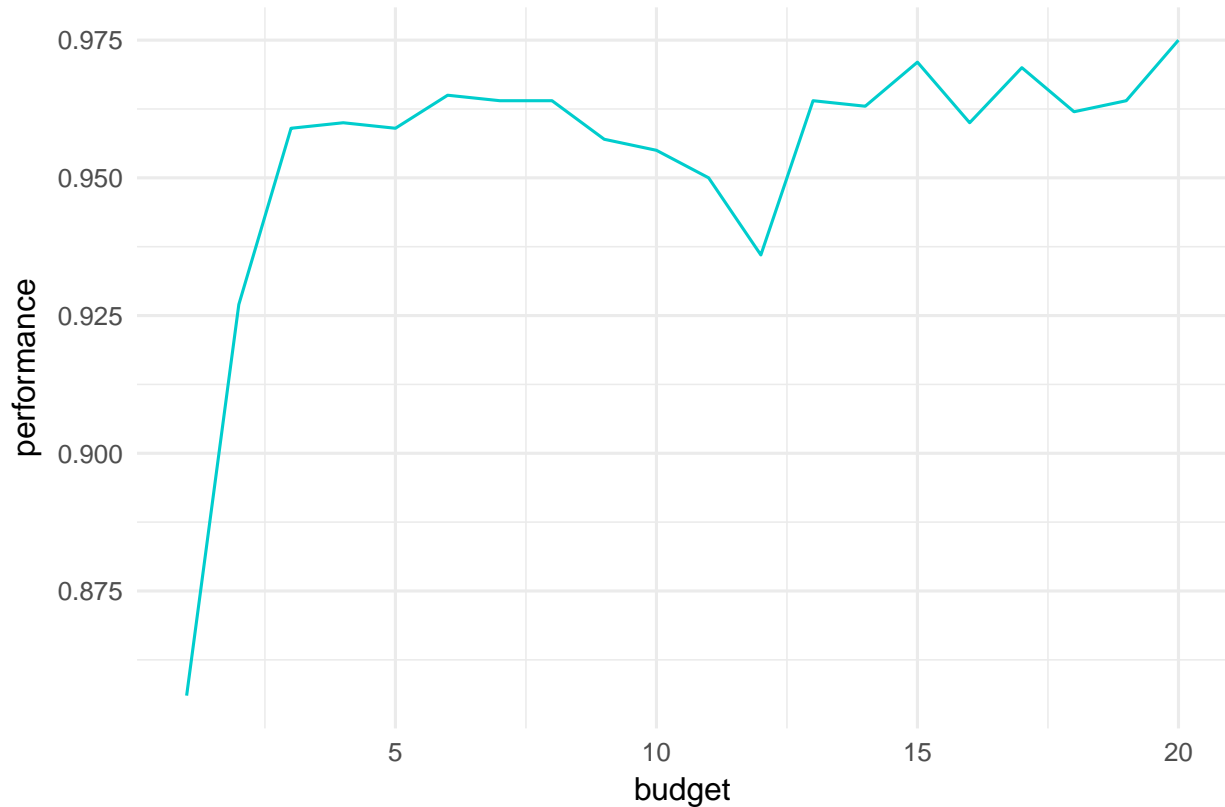


Figure 27:

The `$visPerformance()` method visualizes the algorithm storage. On the x-axis we see the budget and on the y-axis the performance. In this case, we plot the development of the validation accuracy.

4.3.4 Extension of Hyperband with Bayesian Optimization

Recall section 4.3.3. Each bracket has a `bracketStorage`, containing all configurations in that bracket as well as their corresponding performance. At each step of successive halving, we write new lines to the `bracketStorage`. Consequently, configurations which “survived” one step of successive halving occur at least two times in the `bracketStorage`.

When we call `hyperband`, another R6 class called `hyperStorage` is automatically being created. This storage container takes `bracketStorage` objects and concatenates them. Thus, the `hyperStorage` contains information of all configurations over all brackets, which have been computed so far. For instance, if we begin to compute the third bracket, the `hyperStorage` already contains all configurations as well as the respective performances of the first and the second bracket.

So instead of random sampling configurations, we could exploit the information in the `hyperStorage` to propose new configurations in a model based fashion. To this we simply have to adjust our `sample.fun`. One potential implementation could look like this:

```

sample.fun.mbo = function(par.set, n.configs, hyper.storage) {
  # If the hyper.storage is empty, sample from the search space, else use MBO.
  if (dim(hyper.storage)[[1]] == 0) {
    lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])
    # That means, we propose configurations for the second bracket, based on the
    # results from the first bracket. For bracket three, we propose configs based
    # on the results from bracket one and two and so on..
  } else {
    catf("Proposing points")
    ctrl = makeMBOControl(propose.points = n.configs)
    # Set the infill criterion: confidence bound
    ctrl = setMBOControlInfill(ctrl, crit = crit.cb)
    designMBO = data.table(hyper.storage)
    # We have to keep in mind, that some configurations occur multiple
    # times. Here we choose to aggregate their performance according to
    # a rule. For each configuration that occurs more than once:
    # aggregate configs by selecting their best performance.
    designMBO = data.frame(designMBO[, max(y), by = names(configSpace$pars)])
    colnames(designMBO) = colnames(hyper.storage)[-(length(configSpace$pars) + 1)]
    # initSMBO from mlrMBO enables us to conduct human-in-the-loop MBO.
    opt.state = initSMBO(
      par.set = configSpace,
      design = designMBO,
      control = ctrl,
      minimize = FALSE,
      noisy = FALSE)
    # Based on the surrogate model, proposePoints yields us new configurations:
    prop = proposePoints(opt.state)
    propPoints = prop$prop.points
    rownames(propPoints) = c()
    # We want our configs as a list to easily feed them into the mlr learner:
    propPoints = convertRowsToList(propPoints, )
    return(propPoints)
  }
}

```

Here we accessed the mlrMBO package (Bischl et al. 2017) as a toolbox for model-based optimization as well as the ranger package (Wright & Ziegler 2017) to conduct random forest regression. Our acquisition function is qLCB (Hutter et al. 2012) which is an extension of the lower confidence bound.

This basic approach will use the hyperStorage and aggregate multi-occurring configurations by their best performance.

Now we just have to run hyperband with our new `sample.fun`

```
hyperbandrMBO = hyperband(
    problem = problem,
    max.resources = 81,
    prop.discard = 3,
    max.perf = TRUE,
    id = "myModelBasedCNN",
    par.set = configSpace,
    sample.fun = sample.fun.mbo,
    init.fun = init.fun,
    train.fun = train.fun,
    performance.fun = performance.fun)
```

```
## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Proposing points
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Proposing points
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Proposing points
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Proposing points
## Iteration 0, with 1 Algorithms left (Budget: 81)
```

As a result of the Bayesian Optimization procedure, we experience a small overhead, leading to slightly longer computational times.

To compare the results of our vanilla hyperband with those from our model based combination, we plot both hyperVis functions:

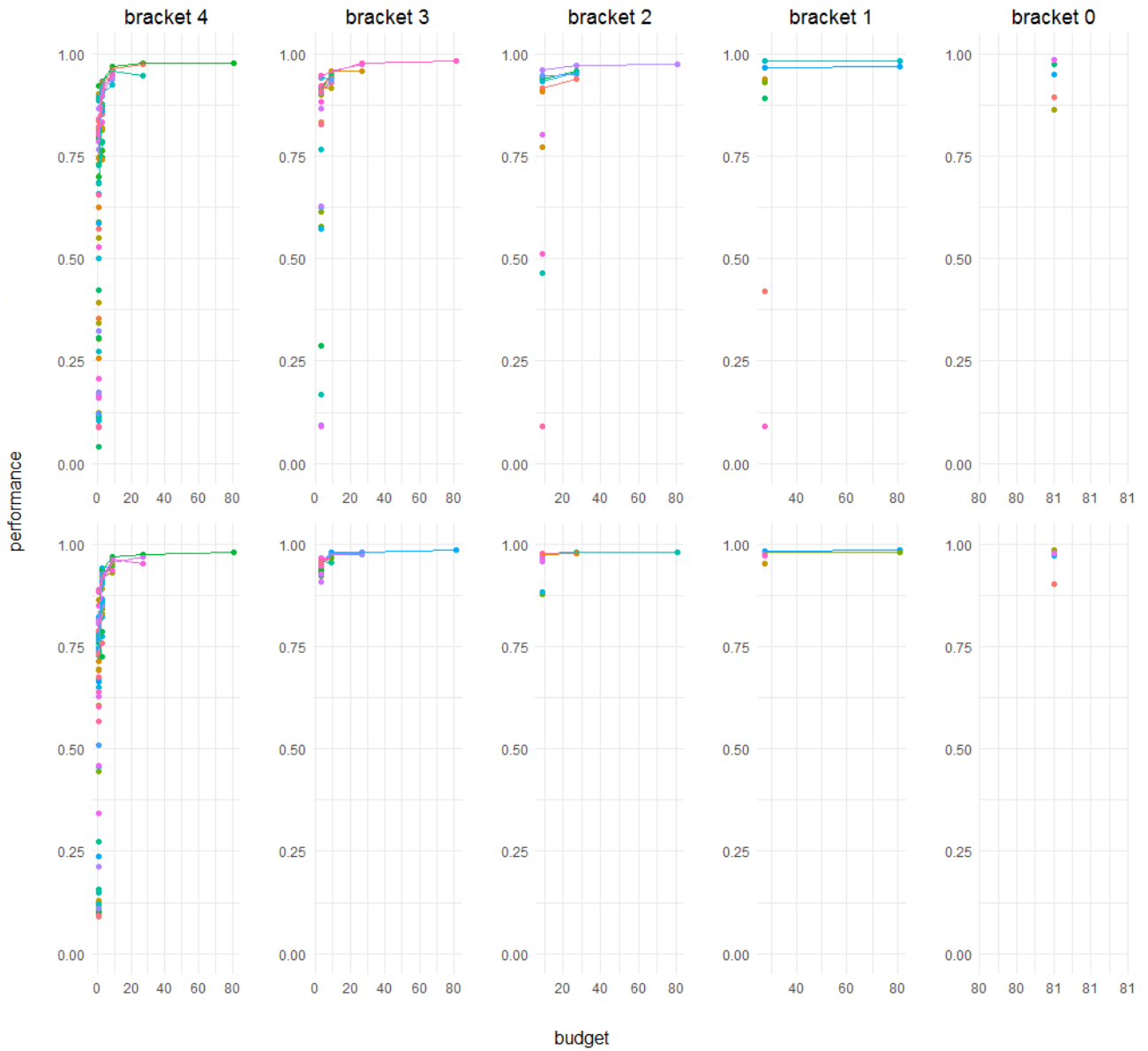


Figure 28: Top row: vanilla hyperband, bottom row: hyperband with model based sampling strategies. On the x-axis, we see the budget and on the y-axis the performance. The model based variant seems to be proposing mainly configurations which exhibit a higher initial performance.

4.3.5 Optimization of a Gradient Boosting Model with Hyperband

Now we would like to optimize a gradient boosting ensemble with hyperbandr.

We opt to utilize XGBoost (Chen et al. 2018) as our framework. To keep things simple, we use the inbuilt agaricus mushroom data and keep the default train/test split.

The train data has 6513 instances and 126 features. Its task is to predict for each of the 1611 test instances, whether the mushroom is poisonous or not, e.g. a binary classification task.

```
# XGBoost requires xgb.DMatrix objects as inputs.
train.set = xgb.DMatrix(agaricus.train$data, label = agaricus.train$label)
test.set = xgb.DMatrix(agaricus.test$data, label = agaricus.test$label)
problem = list(train = train.set, val = test.set)
```

Our `configSpace` comprises the `max_depth` of each tree, the feature subsample ratio `colsample_bytree` and last but not least `subsample`, which controls the ratio of the training samples used to construct a tree.

```
configSpace = makeParamSet(  
  makeIntegerParam("max_depth", lower = 3, upper = 15, default = 3),  
  makeNumericParam("colsample_bytree", lower = 0.3, upper = 1, default = 0.6),  
  makeNumericParam("subsample", lower = 0.3, upper = 1, default = 0.6)  
)
```

We simply use the same `sample.fun` as in chapter 4.3.2.

```
sample.fun = function(par.set, n.configs, ...) {  
  lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])  
}
```

When we initialize an XGBooster, we also set up a watchlist to automatically track the validation error.

```
init.fun = function(r, config, problem) {  
  watchlist = list(eval = problem$val, train = problem$train)  
  capture.output({mod = xgb.train(config, problem$train, nrounds = r, watchlist)})  
  return(mod)  
}
```

XGBoost allows us to pass a pretrained model into the `xgb.train` function.

```
train.fun = function(mod, budget, problem) {  
  watchlist = list(eval = problem$val, train = problem$train)  
  capture.output({mod = xgb.train(xgb_model = mod,  
    nrounds = budget, params = mod$params, problem$train, watchlist)})  
  return(mod)  
}
```

The performance fun just extracts the last entry of our watchlist.

```
performance.fun = function(model, problem) {  
  tail(model$evaluation_log$eval_rmse, n = 1)  
}
```

This time we would like to minimize the root mean square error and thus, set `max.perf` to `FALSE`.

```
hyperboost = hyperband(
  problem = problem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = FALSE,
  id = "xgboost",
  par.set = configSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

Using hyperVis shows us the rmse development accross all brackets.

```
hyperVis(hyperboost)
```

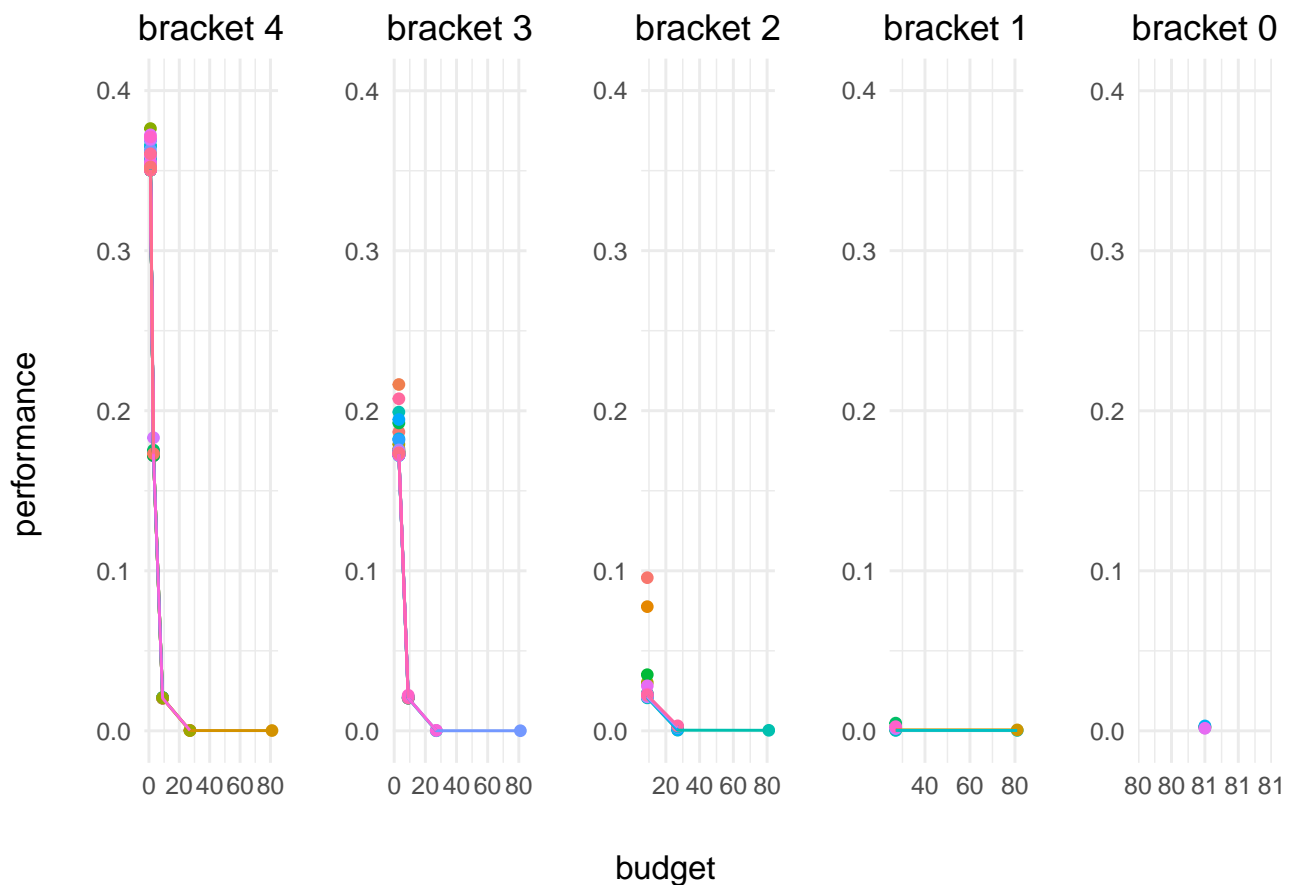


Figure 29: The hyperVis function visualizes each bracket in order to compare them. On the x-axis we see the budget and on the y-axis the performance. Each dot represents one configuration at a certain state. Those dots who are connected by lines are the configurations who were not sorted out immediately but were retrained.

Since the problem was very easy, all five brackets manage to obtain an rmse of almost 0:

```
lapply(hyperboost, function(x) x$getPerformances())
```

```
## [[1]]  
## [1] 5.2e-05  
##  
## [[2]]  
## [1] 4.5e-05  
##  
## [[3]]  
## [1] 0.000354  
##  
## [[4]]  
## [1] 0.000318  
##  
## [[5]]  
## [1] 0.000133
```

4.3.6 Optimization of a Function with Hyperband

Our last demonstration aims towards the optimization of the 2-dimensional branin function. To do this, we deploy the `smoof` package (Bossek 2017).

```
library("smoof")  
braninProb = makeBraninFunction()
```

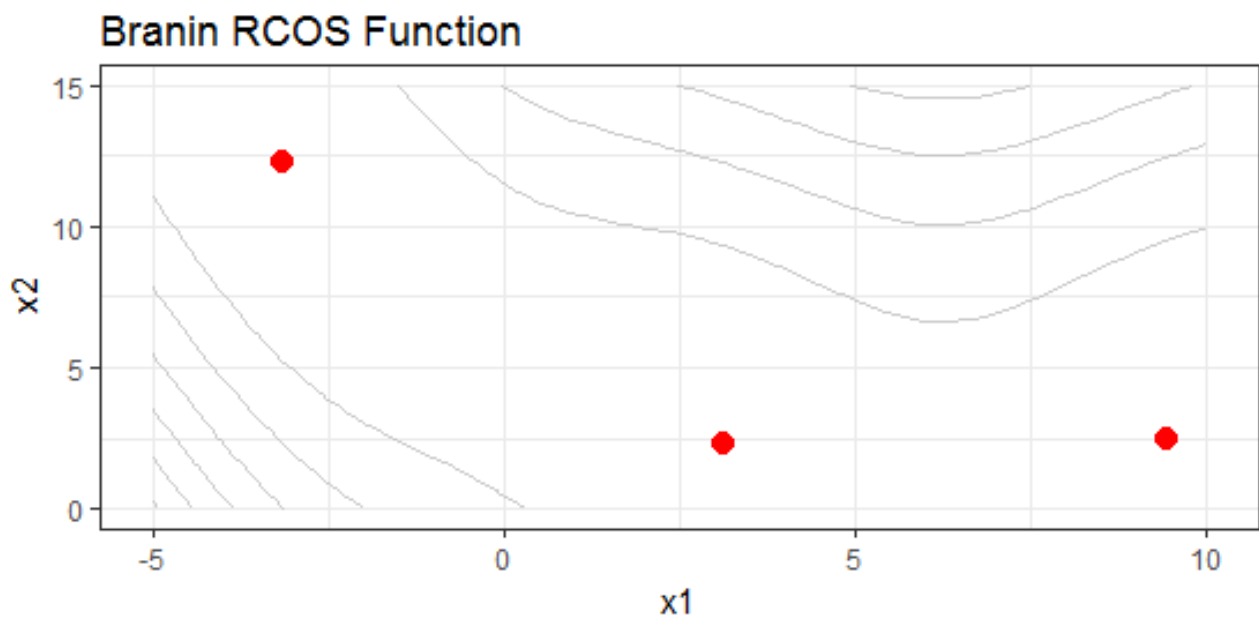


Figure 30: The brain function is a 2-dimensional function and has three global minima (red dots).

```
getParamSet(braninProb)
```

Viable values for x_1 range from roughly -5 to 10.15 .

```
##           Type len Def           Constr Req Tunable Trafo
## x numericvector    2  - -5,0 to 10,15    -    TRUE    -
```

We treat the value of x_1 as our “configuration” and try to find the optimal value for x_2 , our “hyperparameter”. Thus, we want to sample values for x_1 .

```
configSpace = makeParamSet(
  makeNumericParam(id = "x1", lower = -5, upper = 10.1))
```

We can use the same `sample.fun` as we did in chapter 4.3.5 (this time, we don't have to remove NAs)

```
sample.fun = function(par.set, n.configs, ...) {
  sampleValues(par = par.set, n = n.configs)
}
```

The `init.fun` takes a config (e.g. a random value for x_1) and samples a corresponding value for x_2 . Consequently, a model is composed of the tuple (x_1, x_2)

```
init.fun = function(r, config, problem) {
  x1 = unname(unlist(config))
  x2 = runif(1, 0, 15)
  mod = c(x1, x2)
  return(mod)
}
```

To “train” our model, we simply sample values from a normal distribution and add or subtract them from our current x_2 . If the performance improves at that combination of (x_1, x_2) , we keep the point. Otherwise we discard the new x_2 and keep the old one.

This means, for each configuration x_1 , we can only move along the y-axis.

```
train.fun = function(mod, budget, problem) {
  for(i in seq_len(budget)) {
    mod.new = c(mod[[1]], mod[[2]] + rnorm(1, sd = 3))
    if(performance.fun(mod.new) < performance.fun(mod))
      mod = mod.new
  }
  return(mod)
}
```

The `performance.fun` simply evaluates the model at the current values of (x_1, x_2)

```
performance.fun = function(model, problem) {
  braninProb(c(model[[1]], model[[2]]))
}
```

Calling hyperband on this problem will only take 1-2 seconds. As we try to find one of the three global minima, we set `max.perf` to `FALSE`.

```
hyperhyper = hyperband(
  problem = braninProb,
  max.resources = 81,
  prop.discard = 3,
  max.perf = FALSE,
  id = "branin",
  par.set = configSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

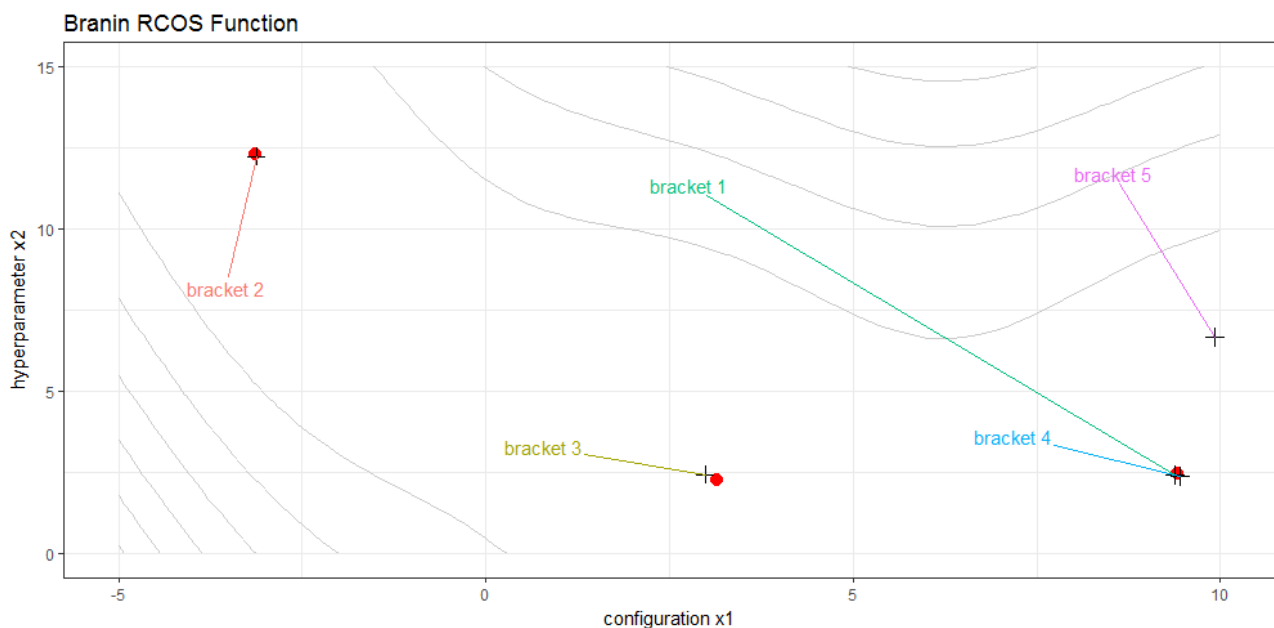


Figure 31: The results of hyperband on the branin function. We see the outcome of our five brackets. Bracket two found the global minimum in the upper left. Bracket three did also find a minimum. Bracket one and two both found the global minimum in the bottom right corner. Only the fifth bracket found none.

5 Benchmark Experiments

In this chapter we will scrutinize several benchmark experiments. We begin to examine the results of Hyperband for XGBoost. Following up, we display the results of two convolutional neural networks.

5.1 XGBoost

The purpose of the first subchapter is to explain the general framework of our benchmark. In the second part, we present the results across several problems.

5.1.1 Experimental Setup

All of the data sets which we used were provided by the OpenML machine learning platform (Vanschoren et al. 2013) and accessed with the OpenML R package (Casalicchio et al. 2017). To elaborate the usability of Hyperband for XGBoost, we used a total of 5 different data sets. Table 4 shows which methods were used on each of them. All problems were benchmarked with Hyperband, a Random Search and “Hyperband + MBO Budget”, the method where the surrogate model tries to learn the spent budget as an additional hyperparameter.

Furthermore, the first two data sets were also tried with the other two combination variants introduced in chapter 3.3. These are “Hyperband + MBO Mean” and “Hyperband + MBO Max”. The first approach aggregates configurations by the mean and the second by the best performance.

Each searcher was tried with different values for R (maximum resources for a single configuration) and η (proportion of configurations to be discarded in each round of Successive Halving).

To be exact, for each data set, we utilized the Cartesian product of

$$\underbrace{\{25, 50, 100, 150, 200, 250\}}_R \times \underbrace{\{2, 3, 4\}}_\eta$$

resulting in a total of 18 different setups for each searcher.

One unit of R corresponds to 5 boosting iterations. For instance, setting $R = 250$ means that a model can be trained for a maximum of $250 \cdot 5 = 1250$ boosting iterations.

Furthermore, we conducted 5 independent replications for each setup. In each of these, we used a different training, validation and testing split. For a data set with n instances, we chose a size of $\frac{2}{3} \cdot n$ for training and $\frac{1}{6} \cdot n$ for validation as well as for the ultimate step of testing. It is important to state that in each replication, all searchers were tried on the exact same split. Thus, overall we performed $18 \cdot 5 = 90$ experiments for every single data set.

Since we only chose classification problems, the performance of each searcher was measured by the accuracy. Therefore, we ensured that each data set had a relatively balanced class distribution.

Table 5 shows the total resources which were spent by each tuple $\{R, \eta\}$. We call these values

$B_{total}(\{R, \eta\})$. Consider the tuple $\{R, \eta\} = \{250, 2\}$. Each searcher was allowed to spend 10.386 resources. A complete run of

$$\{R, 2\}, R \in \{25, 50, 100, 150, 200, 250\}$$

requires 28.832 resources. For $\eta = 2$, all 5 independent replications consumed a total of 149.160 resources. Recall that one unit of resources corresponds to 5 boosting iterations. This means in particular, that for $\eta = 2$, overall $149.160 \cdot 5 = 745.800$ boosting iterations were conducted.

Our goal was to obtain a fair comparison with the Random Search. To achieve this, a Random Searcher was allowed to sample its boosting iterations b_{iters} from $\{1, 2, \dots, R\}$ until

$$\sum b_{iters} = B_{total}(\{R, \eta\})$$

(e.g. until the total budget of the appropriate tuple $\{R, \eta\}$ was exhausted).

Aggregated, this experiment utilized

$$\begin{aligned} \sum_{R, \eta} B_{total}(\{R, \eta\}) &= \left[\underbrace{(149.160 + 83.300 + 51.235)}_{\eta=2} \cdot \underbrace{3}_{\text{datasets}} \cdot \underbrace{3}_{\text{searcher}} \right] \\ &\quad + \left[\underbrace{(149.160 + 83.300 + 51.235)}_{\eta=2} \cdot \underbrace{2}_{\text{datasets}} \cdot \underbrace{5}_{\text{searcher}} \right] \\ &= 5.390.205 \end{aligned}$$

resources or 16.738.005 boosting iterations.

The majority of the experiments were facilitated on the LRZ Linux Cluster CoolMUC-2. It provides Intel Xeon E5-2697 CPUs with 24 cores per node. In addition to this, two personal computers with an i5-4690K and an i7-6700K were utilized continuously. The whole experiment required a total of 14.15 days across all of these machines (run time in CPU hours).

Table 4: XGBoost data sets and corresponding benchmark methods. We used a total of 5 data sets. All of them were benchmarked with Hyperband, a Random Search and Hyperband MBO Budget. In addition to this, the first two data sets were also benchmarked with the two other variants which we discussed in chapter 3.3.

data set	Hyperband	HMBO Budget	HMBO Mean	HMBO Max	Random Search
categorical	x	x	x	x	x
cnae-9	x	x	x	x	x
steel	x	x			x
gesture	x	x			x
letter	x	x			x

Table 5: This table shows the total amount of budget, which results from each tuple of $\{R, \eta\}$. For instance: one run with an arbitrary searcher for the tuple $\{250, 2\}$ spends a total of 10.386 resources. Each searcher conducted five independent replications for all tuples. Thus for $\{R, 2\}$, a total of $29.832 \cdot 5 = 149.160$ units of budget were issued. Since one unit corresponds to 5 boosting iterations, for $\eta = 2$, one searcher performed a total of $149.160 \cdot 5 = 745.800$ boosting iterations.

	R							
η	25	50	100	150	200	250	sum	5 repls
2	434	1.250	3.221	6.232	8.309	10.386	29.832	149.160
3	191	661	1.951	2.927	3.903	7.027	16.660	83.300
4	193	387	1.381	2.071	2.762	3.453	10.247	51.235

Apart from this, we used the R package `batchtools` (Lang et al. 2017) to implement our experiments. This ensures reproducibility of all results as well as clean and readable code. Most importantly, it enabled us to work with the batch system of the LRZ Linux Cluster.

Table 6 shows the search space which was employed. In essence, we used the same setup as the `autoxgboost` R package (Thomas 2017). This package conducts automatic tuning of XGBoost models. Our learning rate was allowed to vary from 0.01 to 0.2. Some regularization parameters, like the λ or the α were transformed logarithmically. The maximum depth of each tree was allowed to vary between 3 and 12. Last but not least we included all three subsample hyperparameters. These involve the fraction of randomly sampled instances for each tree, the fraction of randomly sampled features for each tree and even the fraction of randomly sampled features for each split.

Table 6: Hyperparameter search space for the XGBoost Benchmark. We used the same setup as the `autoxgboost` R package (Thomas 2017). This means that we had a total of 8 hyperparameters. Moreover, the values for γ , λ and α were transformed on a log scale.

Parameter	Range	Trafo	Description
η	[0.01, 0.2]		learning rate
γ	$[-7, 6]$	2^x	minimum loss reduction required to make a split
λ	$[-10, 10]$	2^x	L_2 regularization term on the weights
α	$[-10, 10]$	2^x	L_1 regularization term on the weights
max_depth	$\{3, 4, \dots, 12\}$		Maximum allowed depth of each tree
subsample	[0.5, 1]		fraction of randomly sampled instances for each tree
colsample_bytree	[0.5, 1]		fraction of randomly sampled features for each tree
colsample_bylevel	[0.5, 1]		fraction of randomly sampled features for each split

5.1.2 Results and Evaluation

We try to compare all of our results with those from the corresponding OpenML leaderboards. However, this is not perfectly fair. OpenML tasks use 10-fold cross validation as well as predefined seeds for their splits. In addition to this, they only conduct a training and validation split of proportion $\frac{9}{10}$ to $\frac{1}{10}$. We use $\frac{2}{3}$ for training and consequently, have a more difficult subsampling specification.

Categorical data

Our first data set is called “categorical” (Simonoff 2013). It contains data regarding the DMFT Index (Decayed, Missing, and Filled Teeth) before and after different prevention strategies.

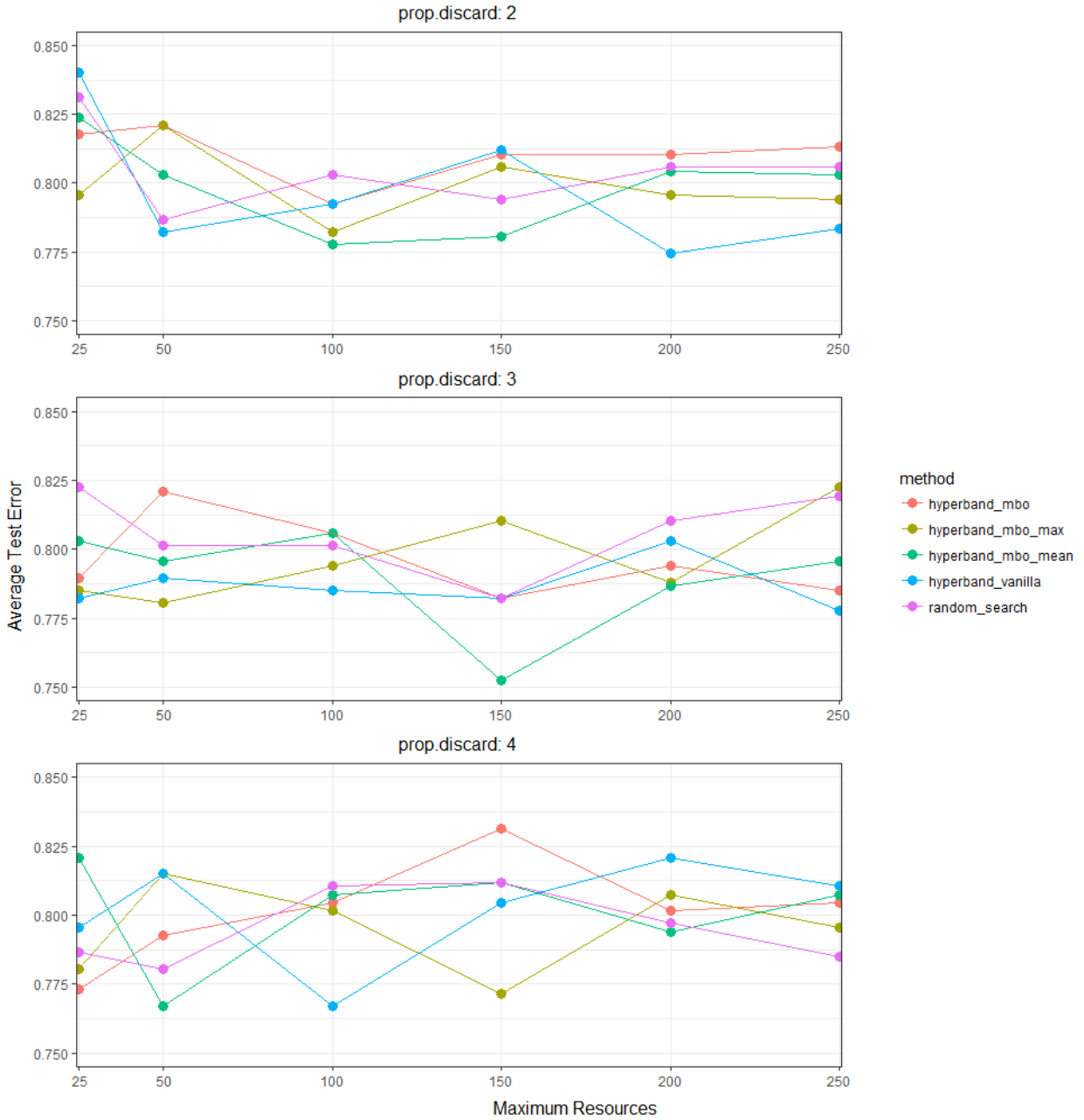


Figure 32: Categorical data: results with $\eta = 2, 3$ and 4. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. As we feed more resources into our searcher, we hardly improve our performance.

The target variable is the prevention strategy and has 6 classes. Besides, the data has 5 features and 787 observations. Out of the 10.093 runs on the OpenML leaderboard, the best algorithm manages to obtain an average test error of (only) 0.7516 (a Random Forest). The best Boosting model yields an average test error of 0.7716.

The top plot in Figure 32 shows the results for $\eta = 2$. All dots represent the average misclassification error of 5 independent replications for the specific searcher. We experience very similar results to those from the OpenML leaderboard. Notably, there is no discernible difference between the behaviour of the three combination variants. This might be due to the problem's extremely high Bayes error.

In the top plot of Figure 33, we see “price” of each searcher’s performance. On the x-axis we can see the average computational time in minutes for one replication.

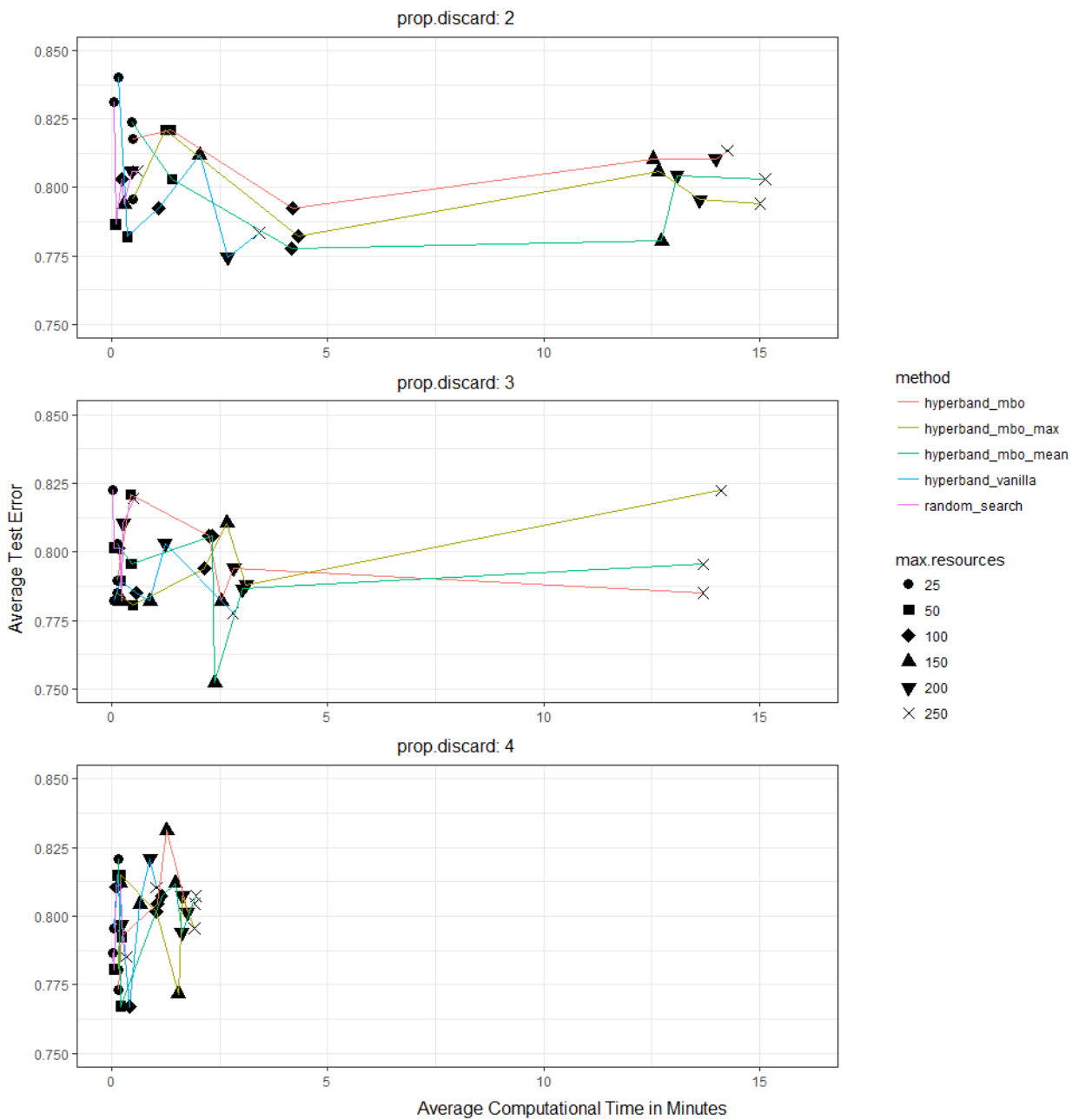


Figure 33: Categorical data: time. On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance.

Compare for instance the crosses, which represent $R = 250$. We can see that the standard Hyperband “hyperband_vanilla” required roughly 3 minutes for one replication (blue line). All three combination variants needed a lot more time to accomplish one replication.

Both plots, Figure 32 and 33 do also show the results for $\eta = 3$ and $\eta = 4$. Overall, the best performance was observed for $\{250, 3\}$ by the “Hyperband + MBO Mean” searcher. It obtains an average test error of 0.7522.

cnae-9 data

Our next problem is called cnae-9. The data set contains 1080 documents of business descriptions for Brazilian companies (Ciarelli & Oliveira 2012). These represent the target of the problem.

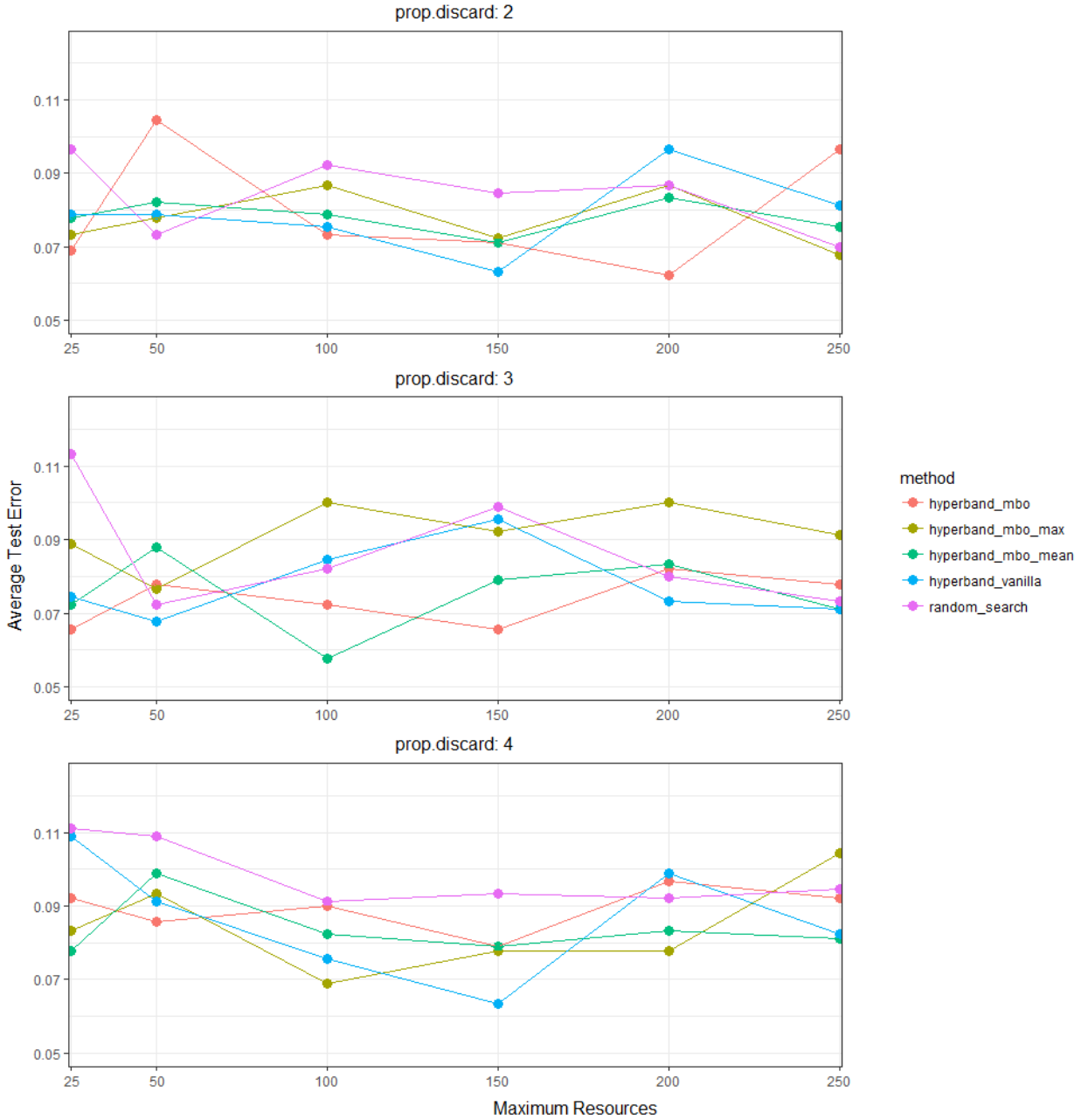


Figure 34: cnae-9: results with $\eta = 2, 3$ and 4. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. As we feed more resources into our searcher, we hardly improve our performance.

This means we have to predict the correct one of 9 possible classes for each business. Furthermore, the data has 857 features, representing the weights of each words frequency in the corresponding document. As of May 2018, there are no trials for this data set at the OpenML website.

The best searcher yields a mean misclassification error of 0.0577. Like before, this is the “Hyperband + MBO Mean” variant, this time for $\{100, 3\}$. Yet all curves are very close together. Figure 35 shows that for $\eta = 2$ and 3, the combination variants require noticeably more time than the vanilla algorithm. Remarkably, the Random Search is able to keep up. Beyond that, it needs much less computational time than all other methods.

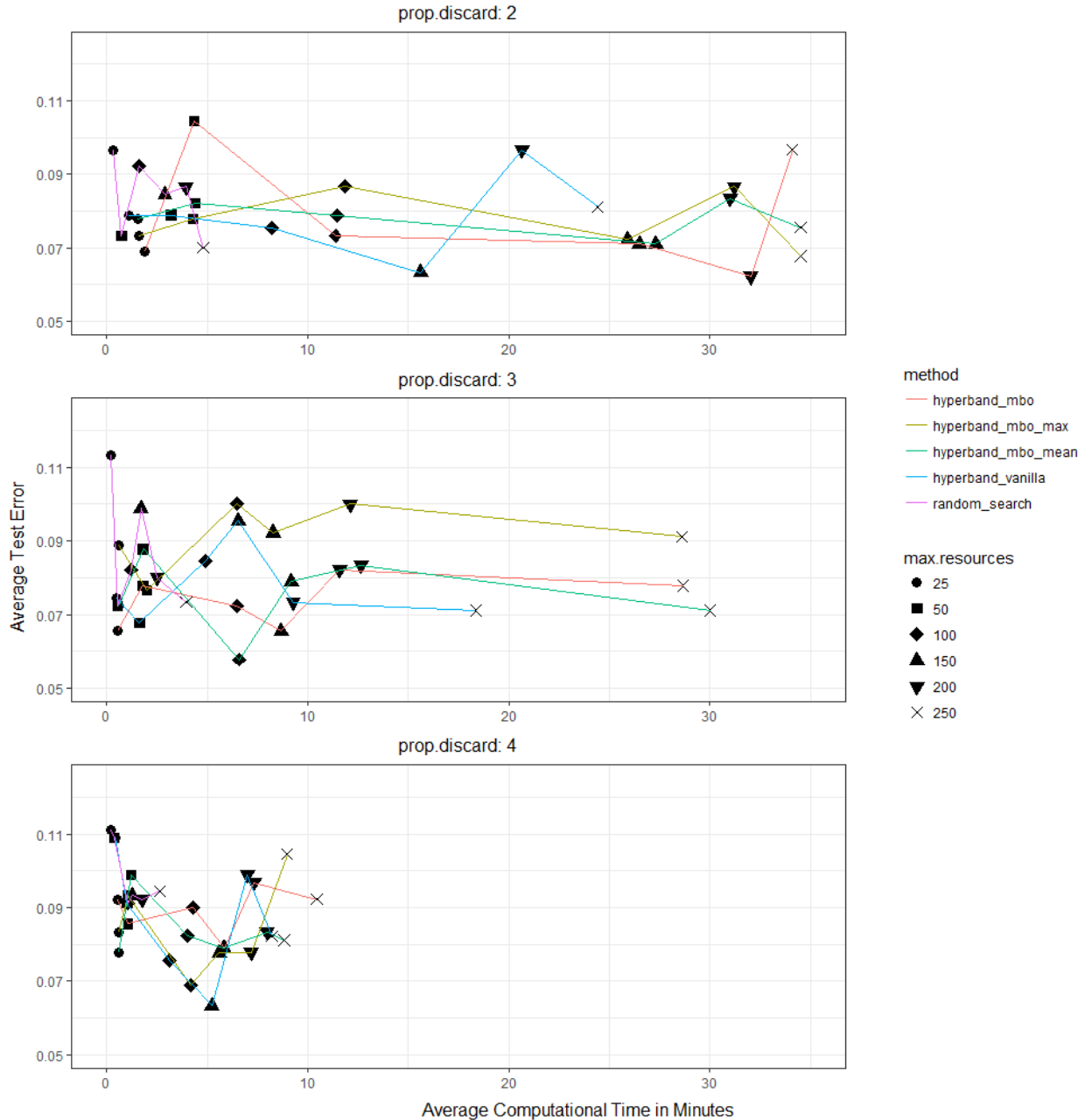


Figure 35: `cnae-9`: time. On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance.

Steel plates fault data

The next data set is called steel plates fault (Buscema & Tastle 2010). Our job is to classify

flaws on steel plates. These include for instance dirtiness, bumps or scratches. Overall, there are 7 unique class labels. We can access 1941 observations and 28 features.

Recall that we now drop “Hyperband + MBO Mean” and “Hyperband + MBO Max” from our experiments (to save time). We chose to keep “Hyperband + MBO Budget” as it represents the most sophisticated variant.

There are 4203 runs on the OpenML leaderboard. The best classifier obtains a mmce of 0.1844 (a Random Forest). Figure 36 reveals that we can challenge this performance. For all three values of η , we find a satisfyingly small mmce. This is 0.1933 for $\{25, 4\}$, 0.1973 for $\{25, 3\}$ and in particular 0.1913 for $\{25, 2\}$.

Another prominent finding is that our combination variant wins at almost every tuple $\{R, \eta\}$.

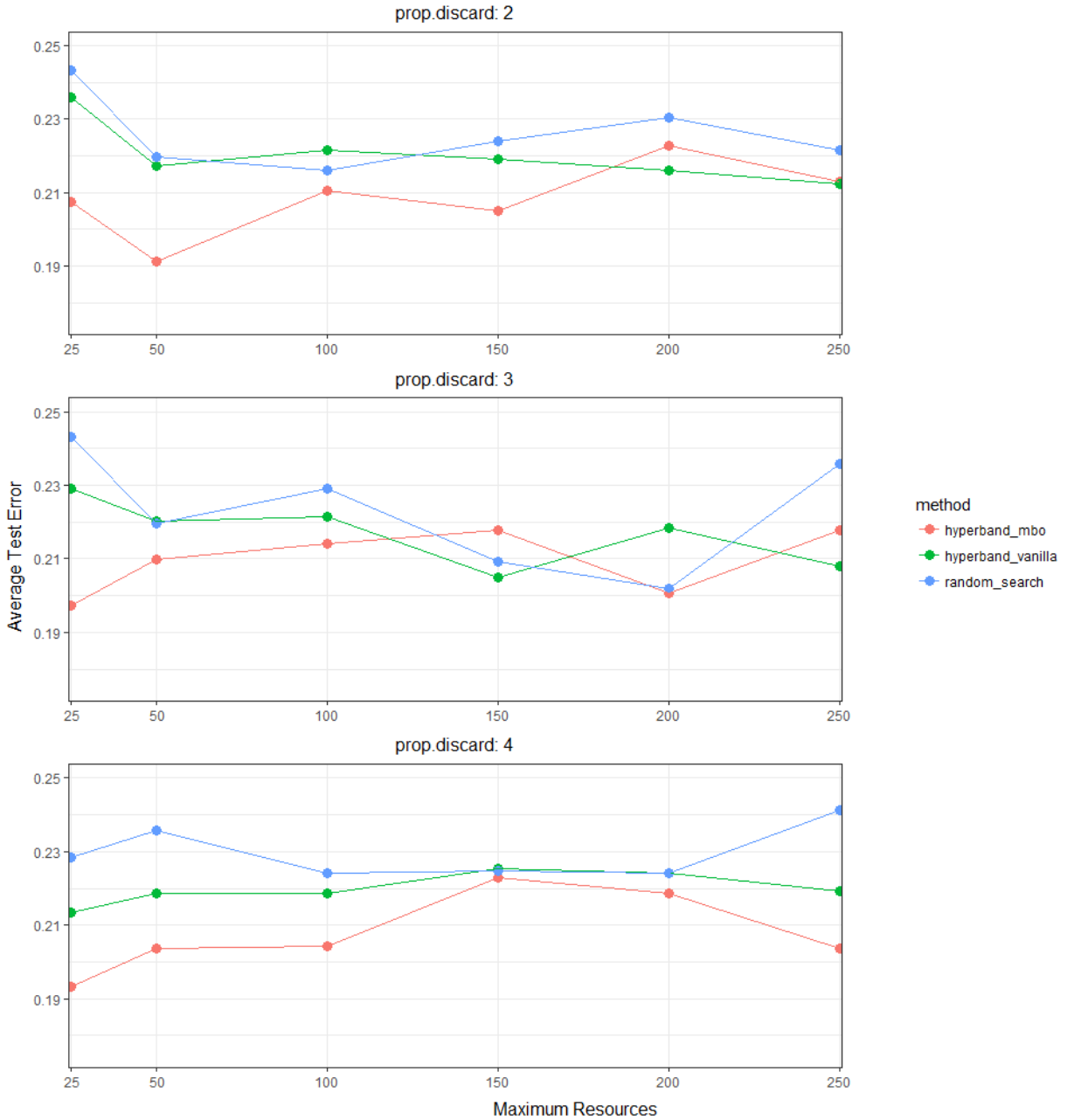


Figure 36: Steel: results with $\eta = 2, 3$ and 4. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. As we feed more resources into our searcher, we hardly improve our performance.

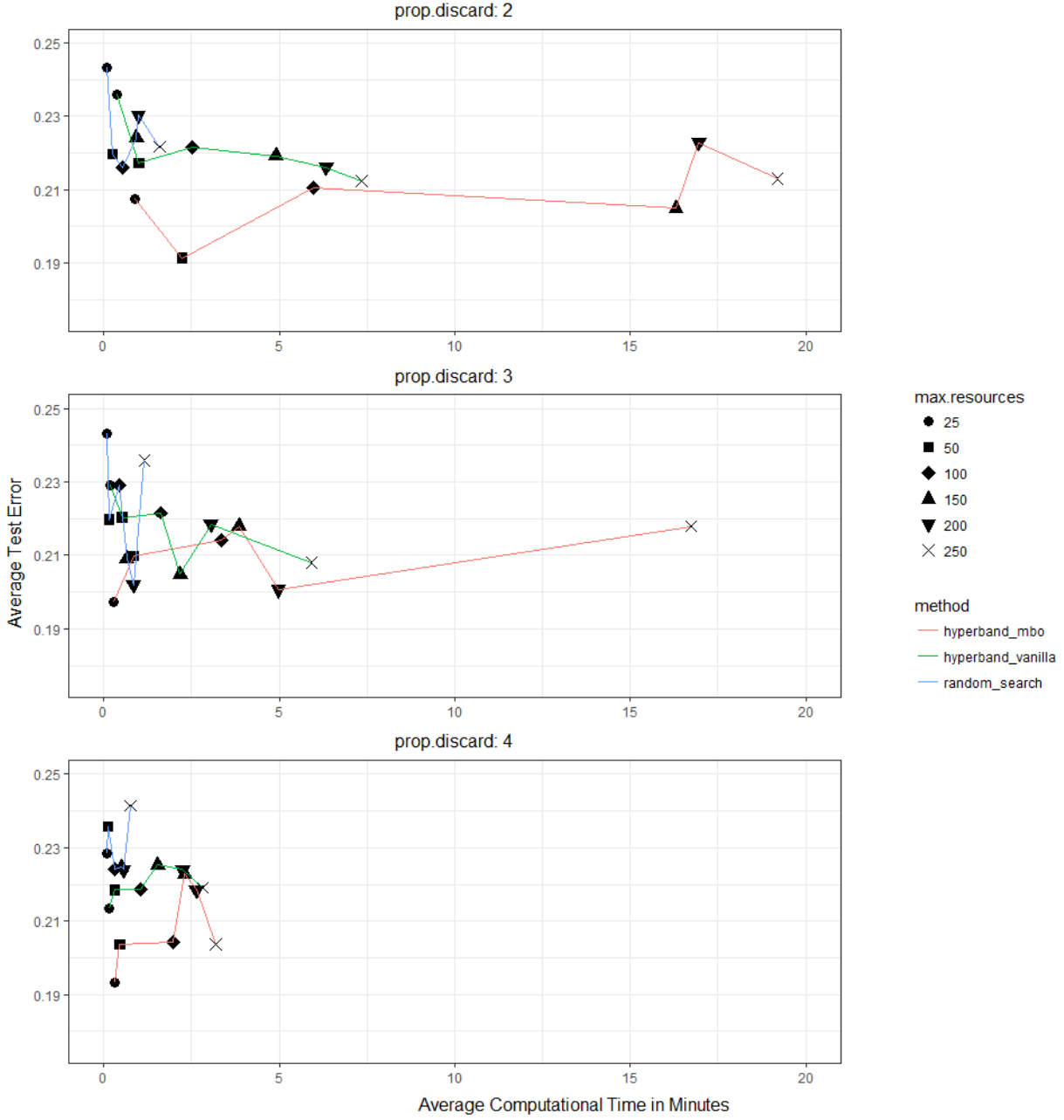


Figure 37: Steel: time. On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance.

In particular for $\eta = 2$ and $\eta = 3$, this good performance comes at the expense of a significantly higher computational time. For instance, one iteration of $\{250, 2\}$ takes the algorithm almost 17 minutes.

Gesture Phase Segmentation data

A slightly bigger data set is represented by the Gesture Phase Segmentation data (Madeo et al. 2013). It has 9873 observations and 33 features. The latter ones are extracted from 7 videos with people gesticulating. They include numeric data, for example the vectorial velocity of the left as well as the right hand across the x, y and z coordinate. These constitute the five classes: “hold”, “rest”, “stroke”, “preparation” and “retraction”.

Once again and out of 7466 runs, the best classifier from the OpenML leaderboard is a Random

Forest (mmce of 0.2789). As can be seen in the top of Figure 38, our best classifier obtains an average test error of 0.3046. This is yielded by the conventional Hyperband for the tuple $\{250, 2\}$.

It seems that the performance might still increase if we feed more resources. Additionally, we see that the performance of the conventional Hyperband and the model-based hybrid almost always obtain similar results. For $\eta = 4$ even the Random Search is able to keep up.

As the size of the data grows, computational times begins to become more substantial (see Figure 39). For $\{250, 2\}$, one replication of “Hyperband + MBO Budget” requires almost 45 minutes. The conventional algorithm barely needs half of this. Not to mention the Random Search, which finishes one replication on average after only 5 minutes.

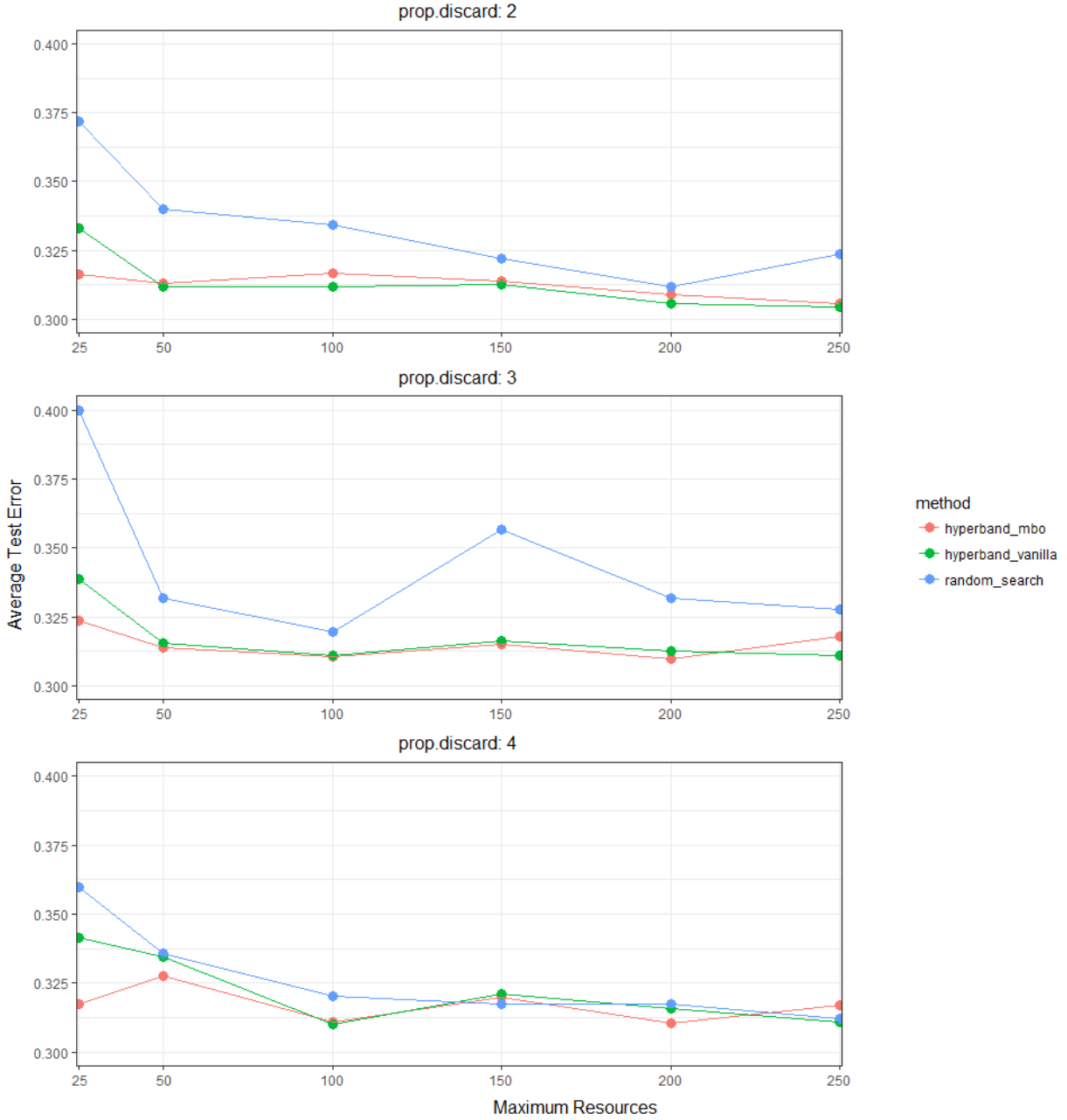


Figure 38: Steel: results with $\eta = 2, 3$ and 4 . On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. As we feed more resources into our searcher, we hardly improve our performance.

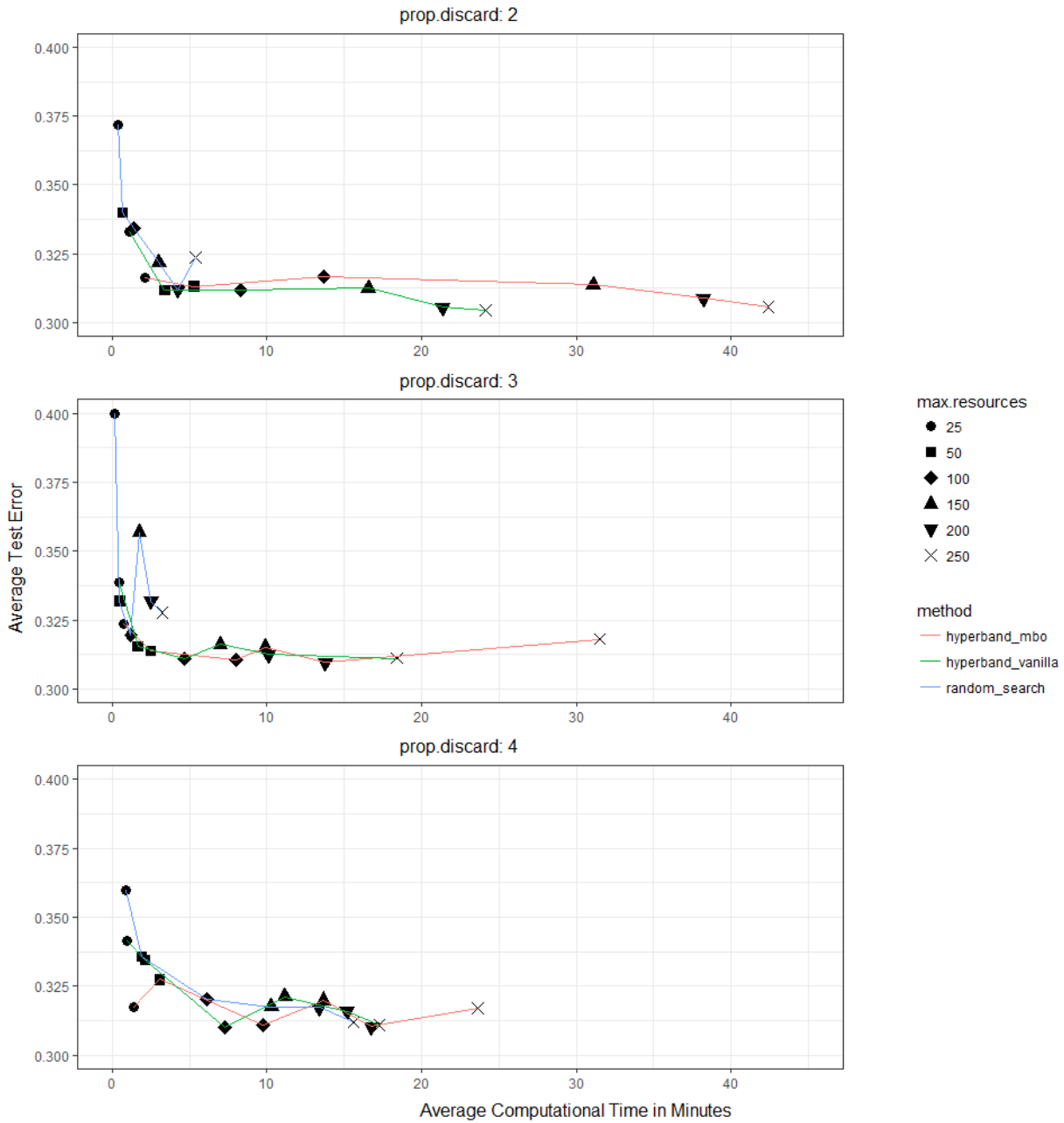


Figure 39: Steel: time. On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance.

Letter data

Our last problem is called letter (Frey & Slate 1991). Its objective is to identify letters on a black-and-white rectangular pixel display, including 20 different fonts. Each instance was converted into a total of 16 numeric features. These cover statistical moments and edge counts. To perform inference, we have 20000 instances available.

There are 22.352 runs on the OpenML leaderboard. Unlike before, this time an SVM manages to obtain an mmce of only 0.0191. The best XGBoost model on this highscore list does only yield 0.2156.

Figure 40 shows that our best searcher achieves a peak performance of 0.0351 (“Hyperband + MBO Budget” at tuple $\{250, 3\}$).

However, both Hyperband variants seem to perform almost identical. This holds for all three

values of η as well as for all values of R . Once again, the Random Search is not really far behind. The computational time graph of Figure 41 raise the question, whether it is worth to spend so much budget for so little increase in performance. For instance, one round with $\{250, 2\}$ takes for both Hyperband variants on average roughly over 3.5 hours.

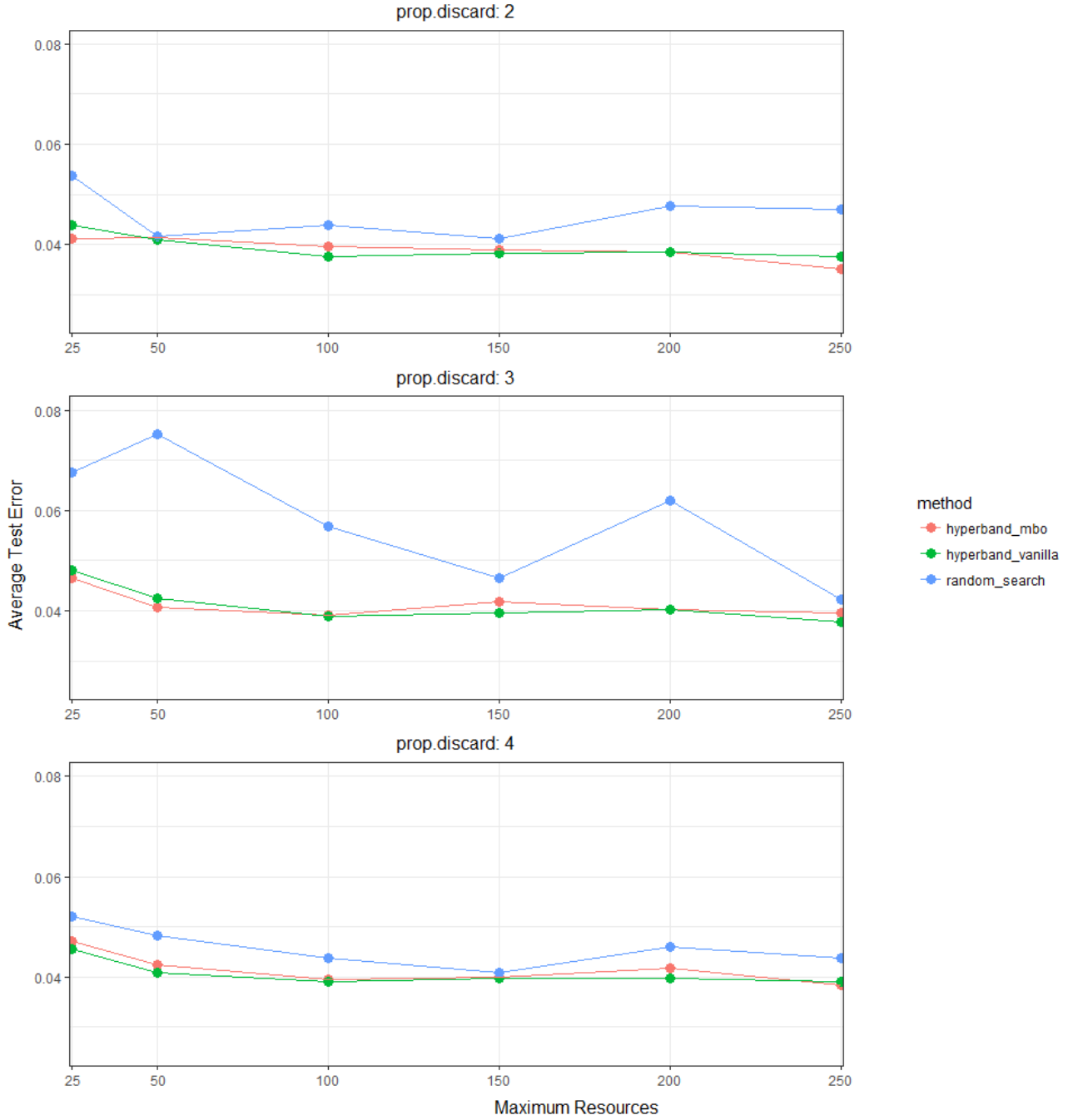


Figure 40: Steel: results with $\eta = 2, 3$ and 4. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. As we feed more resources into our searcher, we hardly improve our performance.

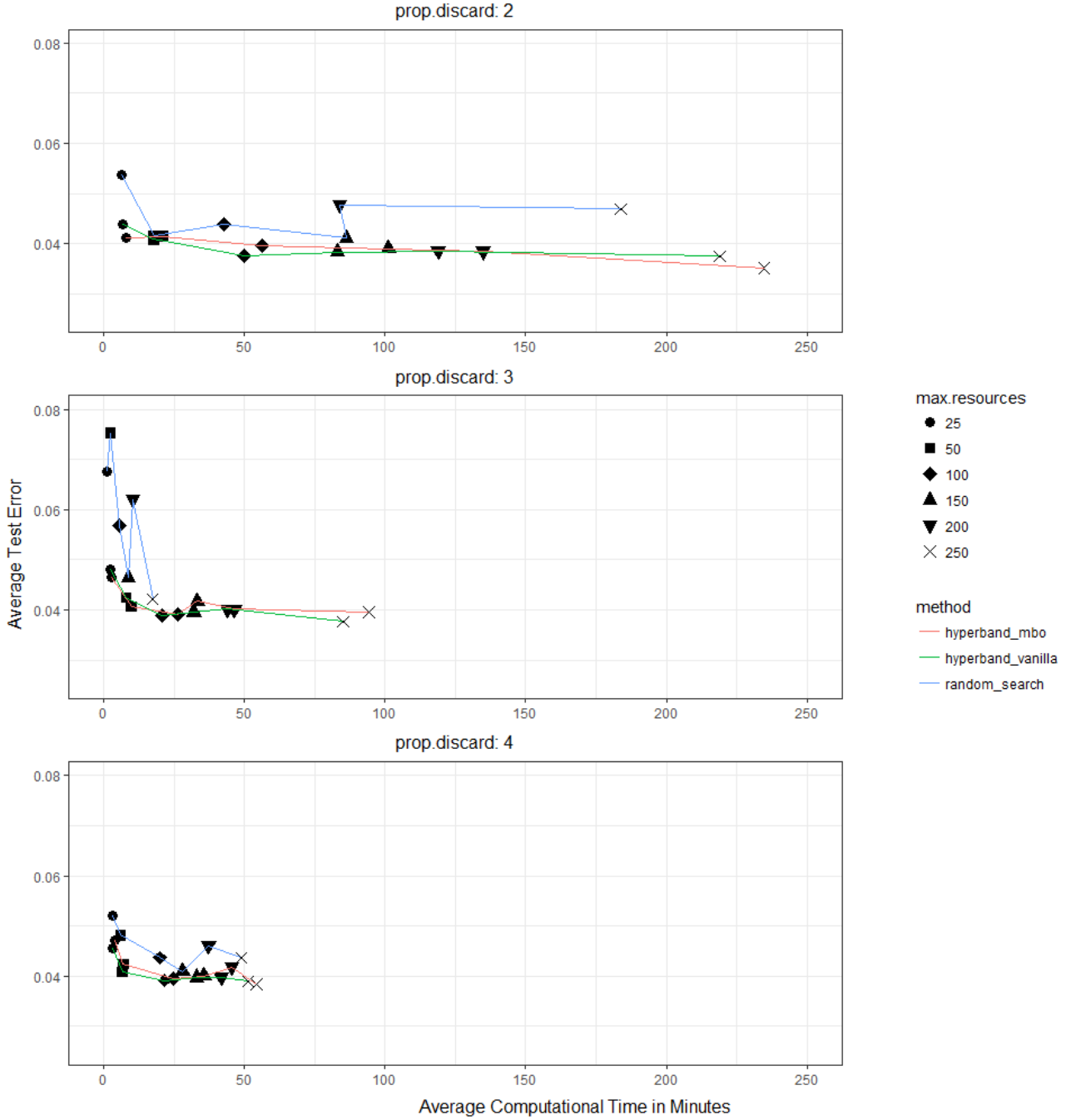


Figure 41: Steel: time. On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance.

Average Ranks

In order to obtain a better overview on the overall performance of our methods, we present an average rank plot (Figure 42).

We see that across all five data sets, “Hyperband + MBO Budget” performed really well for the tuples $\{25, \eta\}$. On average, it ranked 1.13. For the subsequent tuples $\{50, \eta\}$, $\{100, \eta\}$ and $\{150, \eta\}$, there are any barely differences. The two rearmost values for R are being slightly dominated by the conventional Hyperband algorithm. Unsurprisingly, the Random Search is always bottom-placed.

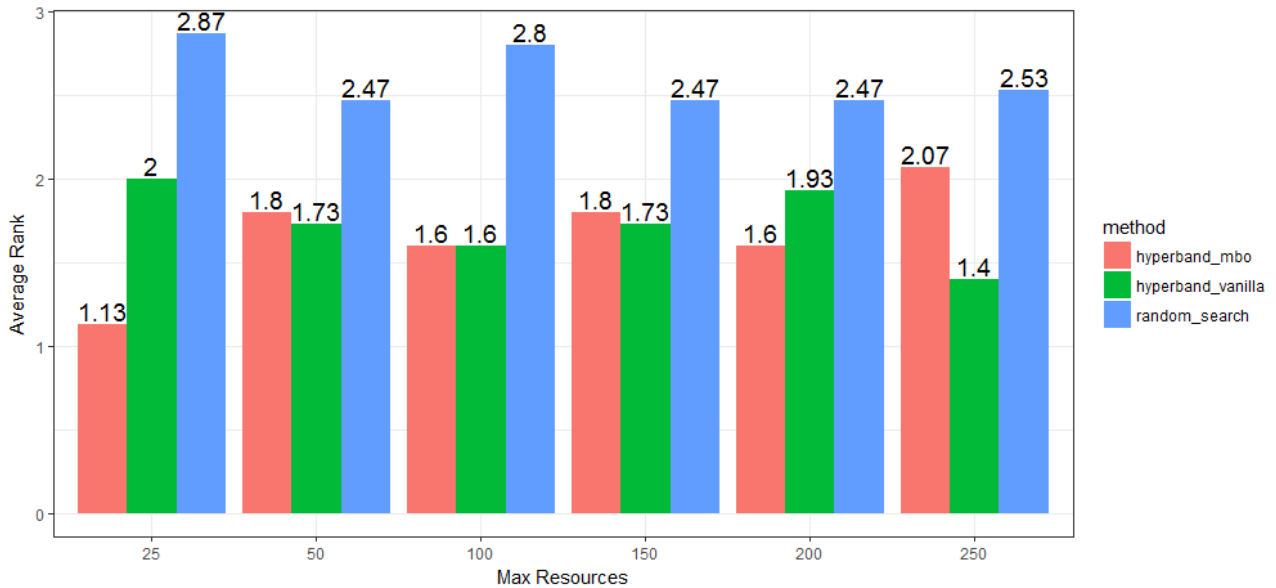


Figure 42: The average rank of our XGBoost Benchmark. Our combination variant did very well for each tuple $\{25, \eta\}$. Later on, the conventional Hyperband algorithm and “Hyperband + MBO Budget” level off. Interestingly, the original Hyperband outperforms both other methods for $R = 250$.

5.2 Convolutional Neural Networks

The second benchmark is dedicated to convolutional neural networks. Like before, we begin by discussing the framework. Afterwards we show results on two problems.

5.2.1 Experimental Setup

To evaluate the effect of our “Hyperband + MBO Budget” variant on neural networks, we try to optimize a CNN on two different data sets:

fashionMNIST (Xiao et al. 2017) and CIFAR10 (Krizhevsky 2009). In addition to this, we utilize the conventional algorithm and a Random Search.

Since neural networks are a lot more expensive than boosting, we only try the tuple:

$$\{R, 4\}, R \in \{25, 50, 100, 150, 200\}$$

This results in a total of 5 different setups for each searcher. One unit of R corresponds to one epoch of training. For fashionMNIST we conduct 5 independent replications for each setup. Due to temporal difficulties, we were only able to run 3 iterations for CIFAR10.

For the latter one, we used the predefined test set with a size of 10.000 observations. The remaining 50.000 instances were partitioned into a random train set of size 40.000 and a random validation set of size 10.000. Different from this strategy, fashionMNIST with overall 70.000 instances was divided into a classic train-val-test split. We used the same rule of chapter 5.1.1. This means $\frac{2}{3}$ for training, $\frac{1}{6}$ for validation and $\frac{1}{6}$ for testing. Summing up, we spent a total of

$$\begin{aligned}
\sum_R B_{total}(\{R, 4\}) &= \left[\underbrace{(193 + 387 + 1.381 + 2.071 + 2.762)}_{R \in \{25, 50, 100, 150, 200\}} \cdot \underbrace{5}_{\text{fashionMNIST}} \right] \\
&\quad + \left[\underbrace{(193 + 387 + 1.381 + 2.071 + 2.762)}_{R \in \{25, 50, 100, 150, 200\}} \cdot \underbrace{3}_{\text{CIFAR10}} \right] \\
&= 54.352
\end{aligned}$$

resources (or computed 54.352 epochs). All experiments were conducted on a single GeForce GTX 1070 and took 4.67 days.

For both benchmarks we used a very similar model architecture as the ones used by Li et al. (2016), Domhan et al. (2015) and Krizhevsky (2009). Table 7 describes the model in detail. The architecture has three convolutional layers, each with filters of size 5×5 . The first layer has

$$\underbrace{5 \cdot 5}_{\text{filter size}} \cdot \underbrace{3}_{\text{RGB channels}} \cdot \underbrace{32}_{\text{feature maps}} + \underbrace{32}_{\text{bias}} = 2.432$$

model parameters. Since we use no dense layers, the model’s overall number of model parameters is 40.426. Each pooling layer uses a step size of two and no padding. Hence each of them halves the height and width of the current feature map.

Table 7: Our CNN model architecture was inspired by cuda-convnet (Krizhevsky 2009). Since we’re dealing with color images, the filter of our first layer has to incorporate the channels of the RGB color model. Hence, each 5×5 filter of layer one has $5^2 \cdot 3 = 75$ parameters. For convolution filters we chose padding and a step size of 1. Thus, both conv layers will conserve its inputs height and width. The pooling filters have a dimension of 2×2 as well as a step size of 2. This will halve the dimension in each pool step. Our architecture has a total of 40.426 parameters (including the bias).

Layer	Filter/neurons	Feature maps	Model parameters	Current dimension
conv 1	$5 \times 5 \times 3$	32	2.432	$32 \times 32 \times 32$
pool 1	2×2			$16 \times 16 \times 32$
conv 2	$5 \times 5 \times 32$	32	9.248	$16 \times 16 \times 32$
pool 2	2×2			$8 \times 8 \times 32$
conv 3	$5 \times 5 \times 32$	64	18.496	$8 \times 8 \times 64$
pool 3	2×2			$4 \times 4 \times 64$
flatten				1×1.024
output	10		10.250	
total			40.426	

Table 8 describes our hyperparameter search space. We opt mainly for regularization parameters, like weight decay, dropout and batch normalization. In addition to this, we include the learning rate and the optimizer. For each optimizer we use the recommended MXNet default values. Domhan et al. (2015) have fewer hyperparameters but also include the number of feature maps into their search space. They allow values from 16 up to 64 feature maps in all of their three layers. This will sometimes greatly increase their architecture size and thus the number of parameters (up to a total of 90.000). Li et al. (2016) on the other hand use a lot more resources. Their minimum value for R is 250 and thereby greater than our maximum value, which is 200. According to them, their benchmark “experiments took the equivalent of over 1 year of GPU hours on NVIDIA GRID K520 cards available on Amazon EC2 g2.8xlarge instances”.

Table 8: Hyperparameter search space for the MXNet Benchmark. We search 10 hyperparameters for our architecture, including the optimizer. For all optimizers we took the recommended default values.

Hyperparameter	Range/Values
learning rate	[0.001, 0.1]
weight decay	[0, 0.01]
dropout input	[0, 0.6]
dropout layer 1	[0, 0.6]
dropout layer 2	[0, 0.6]
dropout layer 3	[0, 0.6]
batch normalization layer 1	{TRUE, FALSE}
batch normalization layer 2	{TRUE, FALSE}
batch normalization layer 3	{TRUE, FALSE}
Optimizer	{SGD, RMSProp, Adam, Adagrad}

Both, Li et al. (2016) and Domhan et al. (2015) applied preprocessing to their data. We use the raw data for our experiments.

5.2.2 Results and Evaluation

fashionMNIST

Our first data set is fashionMNIST (Xiao et al. 2017). The task is to predict the class label of black-and-white images with a dimension of 28×28 . Figure 43 gives an impression on how these images look like. The labels include sneakers, shirts, trousers and so on. We can utilize a total of 70.000 images. These will be split according to the rules of section 5.2.1.

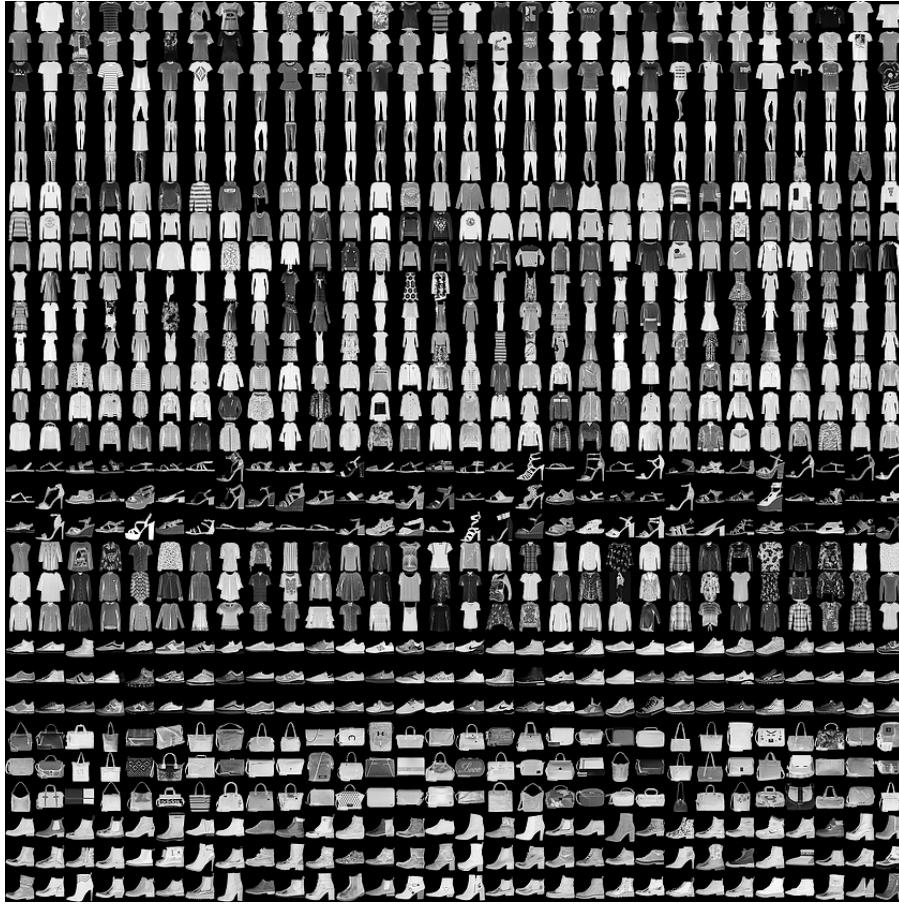
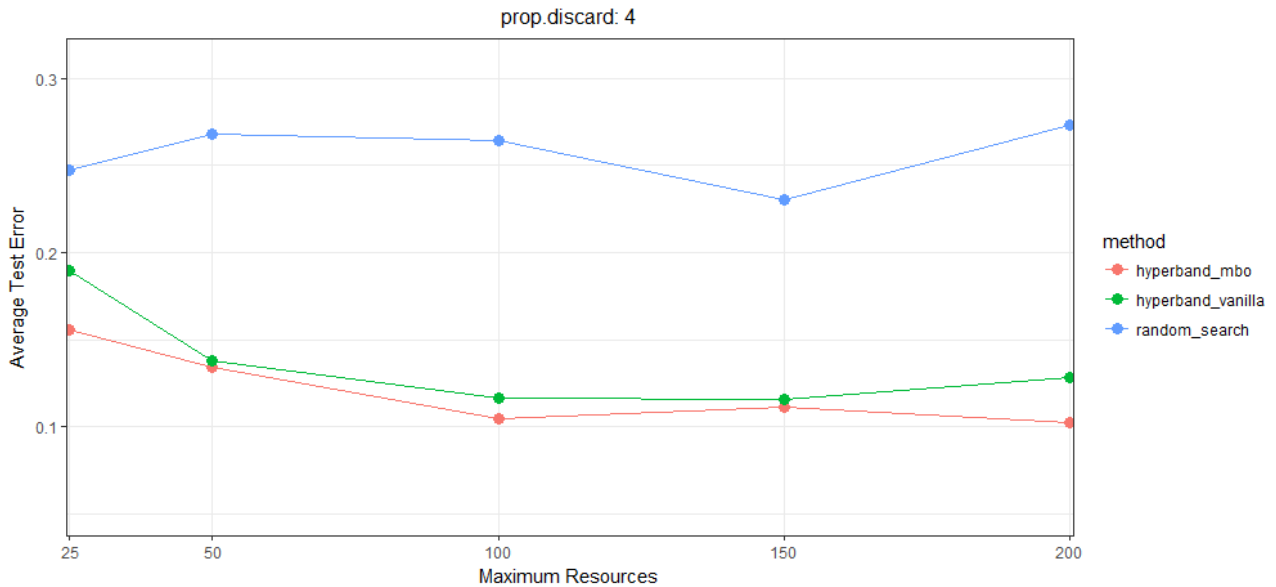
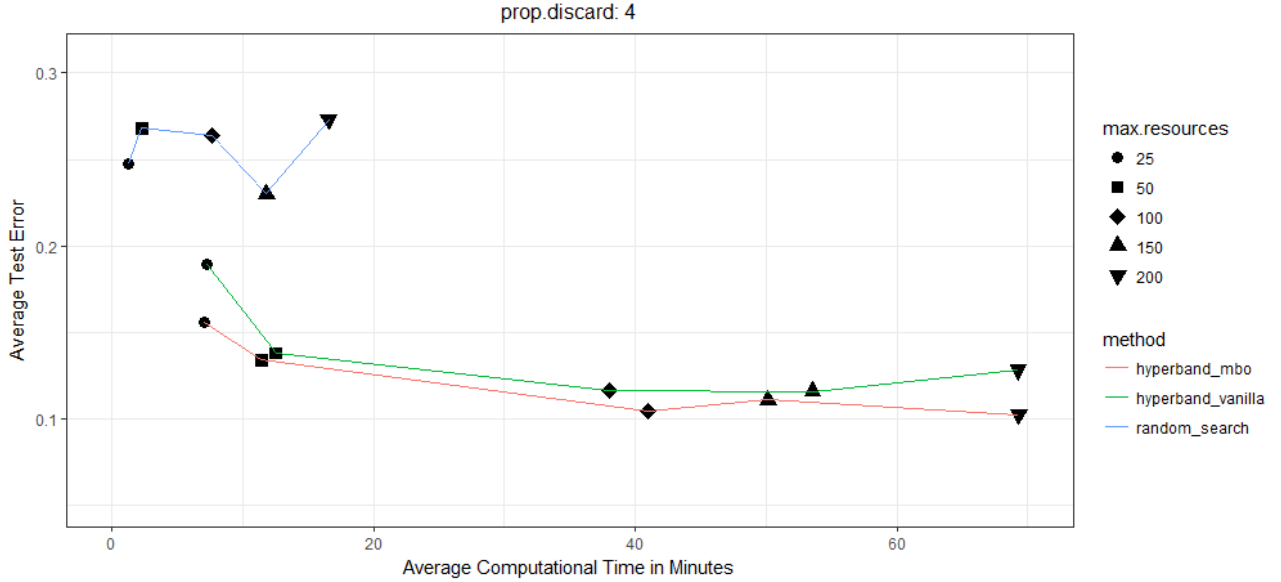


Figure 43: Impression of fashionMNIST. The class labels include sneakers, shirts, trousers and so on. Each image is of size 28×28 (e.g. black-and-white).



(a) fashionMNIST: results with $\eta = 4$. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 5 independent replications. The “Hyperband + MBO Budget” dominates the conventional Hyperband for all values of R . Particularly striking, the Random Search is without any chance to keep pace. For $R = 200$ we observe the lowest average test error or 0.10244.



(b) fashionMNIST: time with $\eta = 4$ On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance. Unsurprisingly, the Random Search is a lot faster than the other two rivals. Unlike what we experienced with XGBoost, the “Hyperband + MBO Budget” does not seem to require more time than the conventional Hyperband.

Figure 44a shows the results of our computations. We immediately see that “Hyperband + MBO Budget” always dominates the conventional algorithm by a small amount. Left behind, the Random Search is no match for both variants of Hyperband. The lowest test error is observed for $R = 200$ and yields 0.10244. This is actually a really good performance on fashionMNIST. Handcrafted algorithms hardly manage to pass the 0.05 error boundary. In addition to this, Figure 44b shows that unlike what we experienced with XGBoost, no computational overhead is noticeable.

CIFAR10

The second and last data set is called CIFAR10 (Krizhevsky 2009). Based on 50.000 color images with size $32 \times 32 \times 3$, our task is to predict the class labels of 10 possible categories. Figure 45 gives an impression on what these images look like. We have airplanes, automobiles, birds, cats and so on.

In Figure 46a we see the results of our computations. Recall that this time, one dot represents only 3 independent replications.

The first impression is very similar to that of fashionMNIST. Across all values for R , “Hyperband + MBO Budget” dominates both, the conventional Hyperband and in particular the Random Search. This time the gap between the two Hyperband variants is even larger. The lowest error of 0.2843 was achieved for $R = 200$.

According to Figure 46b, we see that on average, “Hyperband + MBO Budget” trains even faster than the conventional algorithm. A feasible explanation for this phenomenon might be that the “Hyperband + MBO Budget” variant samples faster configurations.

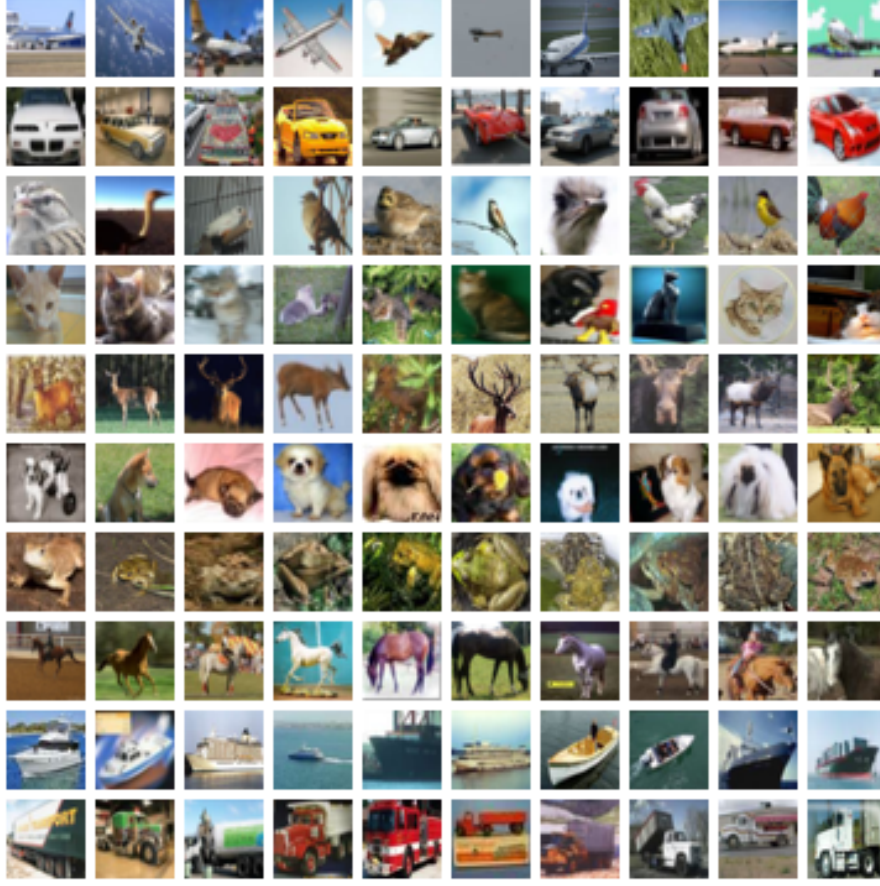
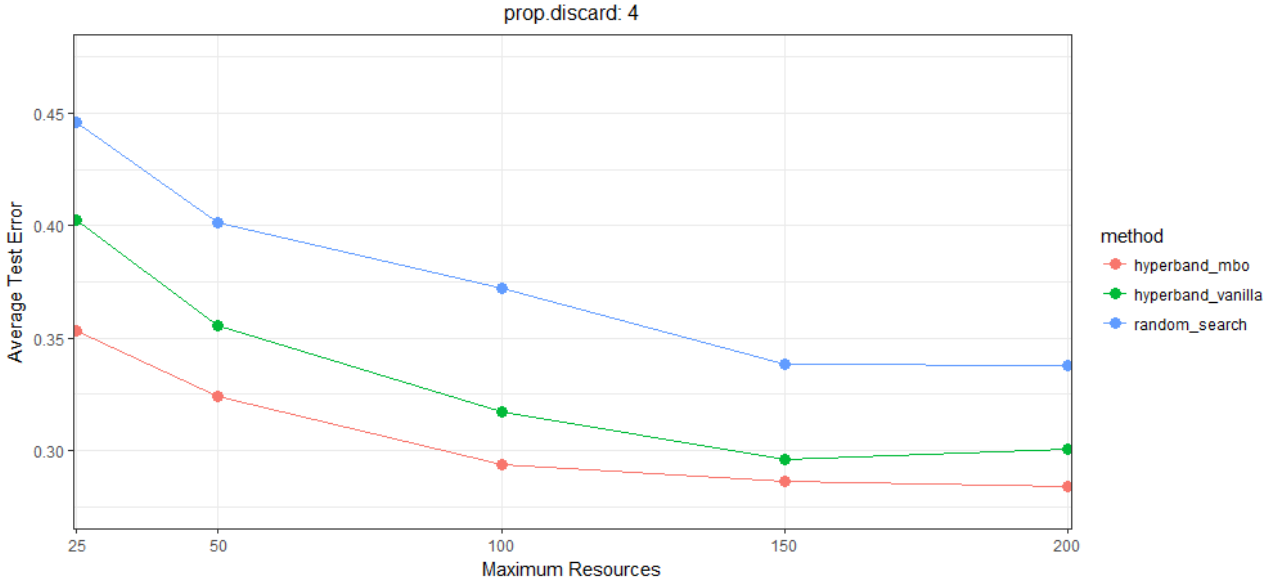
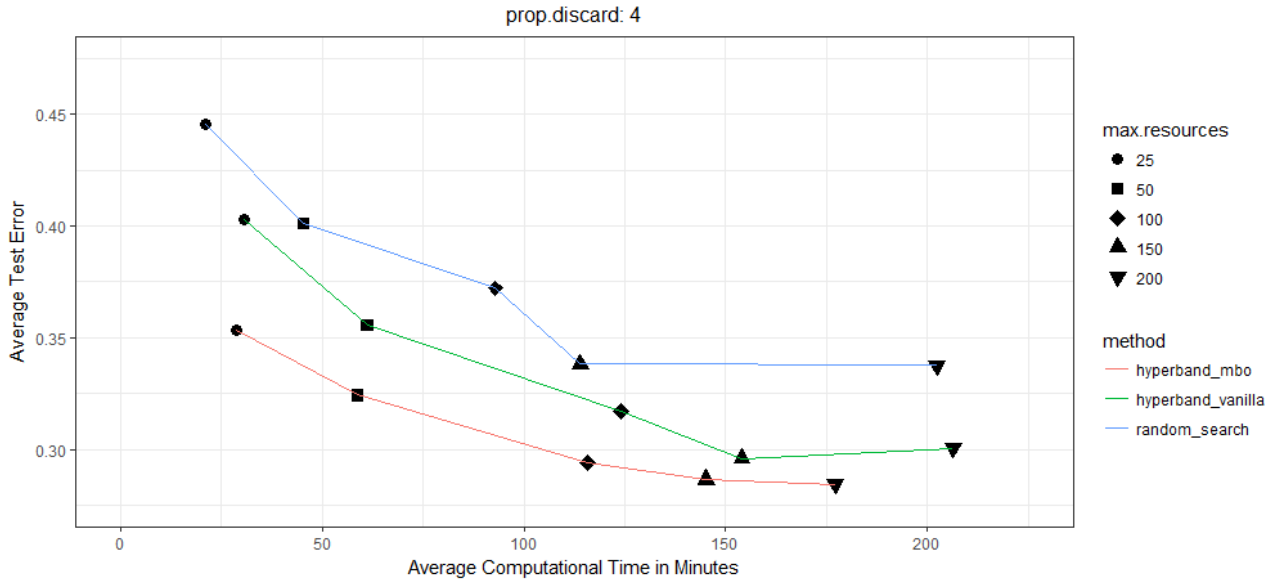


Figure 45: Impression of CIFAR10. The class labels include airplanes, automobiles, birds, cats and so on. Each image is of size $32 \times 32 \times 3$ (e.g. color images).



(a) CIFAR10: results with $\eta = 4$. On the x-axis we see the maximum resources R and on the y-axis the average misclassification error on the test set. Each point represents the average of 3 independent replications. The “Hyperband + MBO Budget” dominates the conventional Hyperband for all values of R . Particularly striking, the Random Search is without any chance to keep pace. For $R = 200$ we observe the lowest average test error at roughly 0.28.



(b) CIFAR10: time with $\eta = 4$ On the x-axis, we can see the average computational time for one replication with each searcher. We can interpret this as the “cost” for the performance. Unlike what we experienced with XGBoost, the “Hyperband + MBO Budget” does not seem to require more time than the conventional Hyperband. In fact, this time it is actually faster.

Average Ranks

For the sake of completeness we do also include the average ranking plot for the neural networks. It comes as no surprise that “Hyperband + MBO Budget” strictly dominates the other two methods and always ranks first.

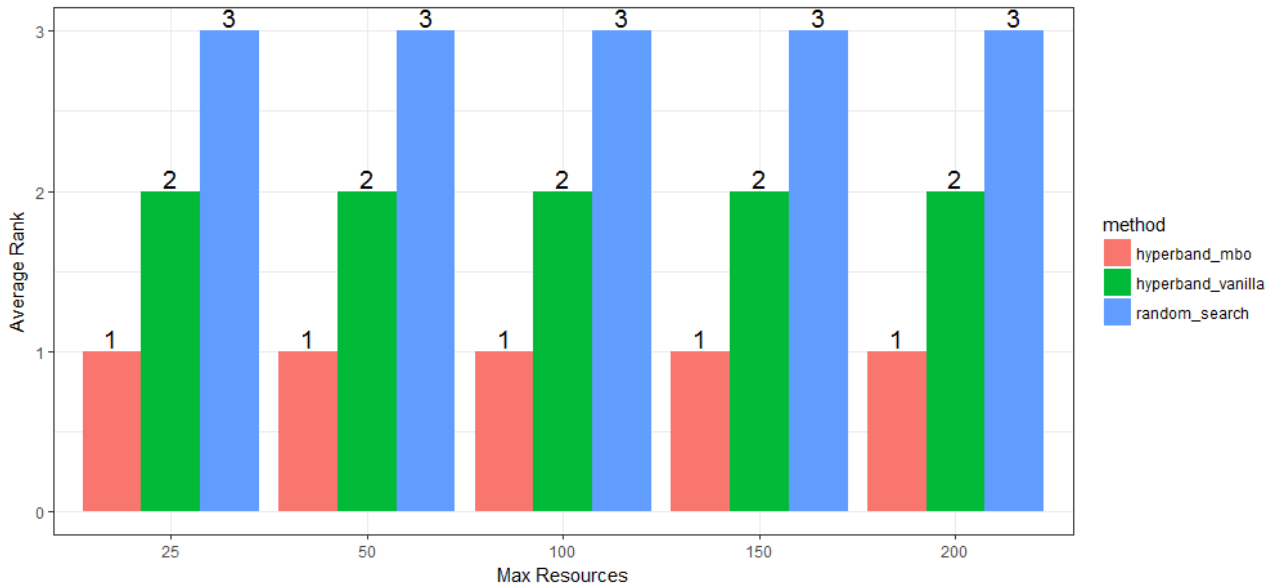


Figure 47

6 Conclusion and Outlook

We approached the difficulties which originate from the optimization of hyperparameters in machine learning models. While there are obviously no shortcuts, some inventive ideas try to tackle these problems.

One of them is model-based optimization. Here we try to find an unknown function which reflects the interaction of our hyperparameters. This strategy has established itself as one of the most successful tuning techniques. Yet Bayesian Optimization comes along with some drawbacks. First of all, it needs many evaluations to obtain a good surrogate surface. The mlrMBO implementation for instance requires $4 \cdot \dim(params)$ samples for its initial design. Consider for instance the tuple $\{R = 200, \eta = 4\}$ with a total budget of 2.762 (see Table 5). For a neural network with 10 hyperparameters, the mlrMBO initialization requires $4 \cdot 10 = 40$ initial evaluations. If we use the same fair sampling technique as we did for the Random Search, we expect to exhaust the total budget before being able to propose a single configuration ($\mathbb{E}(\text{budget}) = \frac{1}{200} \sum_{i=1}^{200} = 100.5, 100.5 \cdot 40 \gg 2.762$). This holds for all tuples $\{R, \eta\}, \eta \in \{3, 4\}$. For this reason, we chose to omit the MBO benchmarking - for now.

In addition to this, Bayesian optimization is generally not very user friendly. It comes with the need for a choice of a suitable surrogate model and in particular an appropriate infill criterion. The latter one requires an optimization itself.

Hyperband approaches the optimization problem from a completely different side. Instead of adaptively proposing new configurations, it allocates resources to more promising ones in an adaptive manner. This procedure is partially inspired by Successive Halving and predominantly intended to tackle the B to n problem (see chapter 3.2.1). Hyperband convinces with its simplicity as well as its good performance. Our experiments suggest also a certain robustness regarding the choice of its input parameters. In particular, the effect of the parameter η seems to be negligible.

Even though Hyperband is already a viable option, we thought it might be profitable to combine the algorithm with Bayesian optimization. We decided to improve the sampling of configurations by transferring information between the brackets. Both Wang et al. (2018) and Falkner et al. (2017) tried to combine Hyperband with Bayesian Optimization. Wang et al. don't utilize this information transfer between brackets, but instead conduct a default MBO run in the first iteration of each bracket. Falkner et al. on the other hand make use of a similar information transfer, but they only partially propose points with MBO and sample the remaining configurations randomly.

For our XGBoost experiments, we found a slight improvement for low input values of R and no noticeable decline for the remainders. When testing with convolutional neural networks, we observed very promising results which need to be verified with a bigger experiment.

A potential problem arises through the fact that all configurations in brackets $\{s_{max} - 1, \dots, 0\}$ are not independent from those in bracket s_{max} . This might violate independence assumptions and thus corrupt the uncertainty estimation of the surrogate model.

Since the hyperbandr package provides a very generic framework, new ideas - including ones to tackle all sorts of problems - can be easily realized.

Another potential improvement could be obtained by replacing Successive Halving with a “[...] statistical approach for selecting the best configurations [...]”(Eiben & Smit 2011) such as iterated F-racing.

It remains to be mentioned that according to the “No free lunch theorem”, there is no optimization technique which is the best for the generic case as well as for all special cases.

List of Figures

1	An neural network start overfitting after roughly 15 epochs.	1
2	A black box uses inputs to generate outputs.	2
3	Human visual cortex and feature extraction	6
4	Sobel-Operator applied to simple black-and-white image	8
5	The difference between predefined filters and random filters.	8
6	Architecture of a small convolutional neural network	9
7	Max pooling procedure.	9
8	The optimizers for a neural network.	11
9	Layers and hyperparameters.	11
10	Randomly erasing parts of input images.	12
11	Grid search, random search and MBO.	14
12	Bayesian optimization of Equation 9.	18
13	LHS vs random sampling.	19
14	Gaussian covariance kernel vs Matérn _{3/2}	20
15	Regression tree on the sinus function of Equation 9.	21
17	The B to n problem.	26
18	Hyperband with $R = 81$ and $\eta = 3$	29
19	Bad prior knowledge in the search space.	32
20	Information transfer between Hyperband brackets.	33
21	The workflow of the hyperbandr package.	36
22	The algorithm object requires seven input.	37
23	The working flow of a bracket object.	38
24	Snippet from the MNIST data that shows 64 different images.	42
25	The hyperVis function visualizes each bracket in order to compare them.	48
26	The \$visPerformance() method for bracket objects.	51
27	The \$visPerformance() method for algorithm objects.	54
28	Vanilla hyperband vs hyperband with model based sampling strategies.	57
29	The hyperVis function.	59
30	The brain function.	60
31	The results of hyperband on the branin function.	62
32	Categorical data: results.	66
33	Categorical data: time	67
34	cnae-9: results.	68
35	cnae-9: time	69
36	Steel: results.	70
37	Steel: time	71
38	Gesture: results.	72
39	Steel: time	73
40	Letter: results.	74

41	Letter: time	75
42	The average rank of our XGBoost Benchmark.	76
43	Impression of fashionMNIST.	79
45	Impression of CIFAR10.	81

List of Tables

1	XGBoost feasible hyperparameters	5
2	Model architecture of Figure 6.	10
3	Example for Successive Halving.	25
4	XGBoost data sets and corresponding benchmark methods.	64
5	XGBoost: how much budget for each tuple $\{R, \eta\}$	65
6	Hyperparameter search space for the XGBoost Benchmark.	65
7	Our CNN model architecture.	77
8	Hyperparameter search space for the MXNet Benchmark.	78

Bibliography

- Bischl, B. (2017), ‘Lecture: Predictive Modeling. Chapter: 11 - Tuning’.
- Bischl, B., Lang, M., Bossek, J., Horn, D., Richter, J. & Kerschke, P. (2017), *ParamHelpers: Helpers for Parameters in Black-Box Optimization, Tuning and Machine Learning*. R package version 1.10
URL: <https://CRAN.R-project.org/package=ParamHelpers>.
- Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G. & Jones, Z. M. (2016), ‘mlr: Machine Learning in R’, *Journal of Machine Learning Research* **17**(170), 1–5.
URL: <http://jmlr.org/papers/v17/15-066.html>.
- Bischl, B., Richter, J., Bossek, J., Horn, D., Thomas, J. & Lang, M. (2017), ‘mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions’.
URL: <http://arxiv.org/abs/1703.03373>.
- Bossek, J. (2017), ‘smoof: Single-and multi-objective optimization test functions.’, *R Journal* **9**(1).
URL: <https://journal.r-project.org/archive/2017/RJ-2017-004/index.html>.
- Breiman, L. (1996), ‘Bagging predictors’, *Mach. Learn.* **24**(2), 123–140.
URL: <http://dx.doi.org/10.1023/A:1018054314350>.
- Breiman, L. (2001), ‘Random forests’, *Machine learning* **45**(1), 5–32.
URL: <https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf>.
- Breiman, L., Friedman, J., Stone, C. J. & Olshen, R. (1984), *Classification and Regression Trees*, The Wadsworth and Brooks-Cole statistics-probability series Wadsworth statistics/probability series, Taylor Francis.
- Buscema, M. & Tastle, W. (2010), A new meta-classifier, in ‘Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American’, IEEE, pp. 1–7.
- Card, D. (2017), ‘The black box metaphor in machine learning’.
URL: <https://towardsdatascience.com/the-black-box-metaphor> (visited on 13.04.2018).
- Casalicchio, G., Bossek, J., Lang, M., Kirchhoff, D., Kerschke, P., Hofner, B., Seibold, H., Vanschoren, J. & Bischl, B. (2017), ‘Openml: An r package to connect to the machine learning platform openml’, *Computational Statistics* **32**(3), 1–15.
URL: <https://link.springer.com/article/10.1007%2Fs00180-017-0742-2>.
- Chang, W. (2017), *R6: Classes with Reference Semantics*. R package version 2.2.2
URL: <https://CRAN.R-project.org/package=R6>.

- Chen, T. & Guestrin, C. (2016), ‘Xgboost: A scalable tree boosting system’, *CoRR* abs/1603.02754.
URL: <http://arxiv.org/abs/1603.02754>.
- Chen, T., He, T., Benesty, M., Khotilovich, V. & Tang, Y. (2018), *xgboost: Extreme Gradient Boosting*. R package version 0.6.4.1
URL: <https://CRAN.R-project.org/package=xgboost>.
- Chen, T., Kou, Q. & He, T. (2017), *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. R package version 0.10.1
URL: <https://github.com/dmlc/mxnet/tree/master/R-package>.
- Chollet, F. (2015), ‘Keras’.
URL: <https://keras.io>.
- Ciarelli, P. M. & Oliveira, E. (2012), ‘cnae-9’.
URL: <https://archive.ics.uci.edu/ml/datasets/CNAE-9>.
- Domhan, T., Springenberg, J. T. & Hutter, F. (2015), Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves., *in* ‘IJCAI’, Vol. 15, pp. 3460–8.
URL: http://ml.informatik.uni-freiburg.de/papers/15-IJCAI-Extrapolation_of_Learning_Curves.pdf.
- Eiben, A. E. & Smit, S. K. (2011), Evolutionary algorithm parameters and methods to tune them, *in* ‘Autonomous search’, Springer, pp. 15–36.
- Falkner, S., Klein, A. & Hutter, F. (2017), ‘Combining hyperband and bayesian optimization’.
- Frey, P. W. & Slate, D. J. (1991), ‘Letter recognition using holland-style adaptive classifiers’, *Machine learning* **6**(2), 161–182.
- Fridman, L. (2018), ‘Deep Learning for Self-Driving Cars’.
URL: <https://selfdrivingcars.mit.edu/> (visited on 26.03.2018).
- Friedman, J. H. (2001), ‘Greedy function approximation: a gradient boosting machine’, *Annals of statistics* pp. 1189–1232.
URL: <https://statweb.stanford.edu/~jhf/ftp/trebst.pdf>.
- Glorot, X. & Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* ‘Proceedings of the thirteenth international conference on artificial intelligence and statistics’, pp. 249–256.
URL: <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.
- Hastie, T., Tibshirani, R. & Friedman, J. (2009a), Additive models, trees, and related methods, *in* ‘The Elements of Statistical Learning’, Springer Series in Statistics, Springer New York,

pp. 295–336.

URL: http://dx.doi.org/10.1007/978-0-387-84858-7_9.

Hastie, T., Tibshirani, R. & Friedman, J. (2009b), Random forests, *in* ‘The Elements of Statistical Learning’, Springer Series in Statistics, Springer New York, pp. 587–604.

URL: http://dx.doi.org/10.1007/978-0-387-84858-7_15.

He, K., Zhang, X., Ren, S. & Sun, J. (2015), ‘Delving deep into rectifiers: Surpassing human-level performance on imagenet classification’, *CoRR* **abs/1502.01852**.

URL: <http://arxiv.org/abs/1502.01852>.

Herzog, M. H. & Clarke, A. M. (2014), ‘Why vision is not both hierarchical and feedforward’.

URL: <https://www.scienceopen.com/document?id=2d184a64-d52e-42b4-9acf-25ba1e637a38>.

Hutter, F., Hoos, H. H. & Leyton-Brown, K. (2012), Parallel Algorithm Configuration, *in* ‘Proceedings of the Learning and Intelligent Optimization Conference LION 6’, pp. 55–70.

Ioffe, S. & Szegedy, C. (2015), ‘Batch normalization: Accelerating deep network training by reducing internal covariate shift’, *CoRR* **abs/1502.03167**.

URL: <http://arxiv.org/abs/1502.03167>.

Jamieson, K. G. & Talwalkar, A. (2015), ‘Non-stochastic best arm identification and hyperparameter optimization’, *CoRR* **abs/1502.07943**.

URL: <http://arxiv.org/abs/1502.07943>.

Jones, D. R., Schonlau, M. & Welch, W. J. (1998), ‘Efficient global optimization of expensive black-box functions’, *Journal of Global optimization* **13**(4), 455–492.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. & Tang, P. T. P. (2016), ‘On large-batch training for deep learning: Generalization gap and sharp minima’, *CoRR* **abs/1609.04836**.

URL: <http://arxiv.org/abs/1609.04836>.

Krizhevsky, A. (2009), Learning multiple layers of features from tiny images, Technical report.

URL: <https://www.cs.toronto.edu/%7Ekriz/learning-features-2009-TR.pdf>.

Lang, M., Bischl, B. & Surmann, D. (2017), ‘batchtools: Tools for r to work on batch systems’, *The Journal of Open Source Software* **2**(10).

URL: <https://doi.org/10.21105/joss.00135>,.

LeCun, Y., Cortes, C. & Burges, C. (2010), ‘Mnist handwritten digit database’, *AT&T Labs* **2**.

URL: <http://yann.lecun.com/exdb/mnist/>.

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A. & Talwalkar, A. (2016), ‘Efficient hyperparameter optimization and infinitely many armed bandits’, *CoRR* **abs/1603.06560**.

URL: <http://arxiv.org/abs/1603.06560>.

- Maas, A. L., Hannun, A. Y. & Ng, A. Y. (2013), Rectifier nonlinearities improve neural network acoustic models, *in* ‘in ICML Workshop on Deep Learning for Audio, Speech and Language Processing’.
URL: https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.
- Madeo, R. C. B., Lima, C. A. M. & Peres, S. M. (2013), Gesture unit segmentation using support vector machines: Segmenting gestures from rest positions, *in* ‘Proceedings of the 28th Annual ACM Symposium on Applied Computing’, SAC ’13, ACM, New York, NY, USA, pp. 46–52.
URL: <http://doi.acm.org/10.1145/2480362.2480373>
- Morar, M., Knowles, J. & Sampaio, S. (2017), *Initialization of Bayesian Optimization Viewed as Part of a Larger Algorithm Portfolio*.
URL: http://ds-o.org/images/Workshop_papers/Morar.pdf.
- Qayyum, A., Anwar, S. M., Majid, M., Awais, M. & Alnowami, M. R. (2017), ‘Medical image analysis using convolutional neural networks: A review’, *CoRR* **abs/1709.02250**.
URL: <http://arxiv.org/abs/1709.02250>.
- Roustant, O., Ginsbourger, D. & Deville, Y. (2012), ‘DiceKriging, DiceOptim: Two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization’, *Journal of Statistical Software* **51**(1), 1–55.
URL: <http://www.jstatsoft.org/v51/i01/>.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. et al. (2015), ‘Imagenet large scale visual recognition challenge’, *International Journal of Computer Vision* **115**(3), 211–252.
URL: <http://www.image-net.org/>.
- Simonoff, J. S. (2013), *Analyzing categorical data*, Springer Science & Business Media.
- Smith, S. L., Kindermans, P. & Le, Q. V. (2017), ‘Don’t decay the learning rate, increase the batch size’, *CoRR* **abs/1711.00489**.
URL: <http://arxiv.org/abs/1711.00489>.
- Snoek, J., Larochelle, H. & Adams, R. P. (2012), Practical bayesian optimization of machine learning algorithms, *in* F. Pereira, C. J. C. Burges, L. Bottou & K. Q. Weinberger, eds, ‘Advances in Neural Information Processing Systems 25’, Curran Associates, Inc., pp. 2951–2959.
URL: <https://arxiv.org/pdf/1206.2944.pdf>.
- Sobel, I. (1968), *Sobel-Operator* — *Wikipedia, The Free Encyclopedia*.
URL: https://en.wikipedia.org/wiki/Sobel_operator (visited on 20.04.2018).

- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), ‘Dropout: A simple way to prevent neural networks from overfitting’, *J. Mach. Learn. Res.* **15**(1), 1929–1958.
URL: <http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf>.
- Strom, N. (2015), Scalable distributed dnn training using commodity gpu cloud computing, *in* ‘Sixteenth Annual Conference of the International Speech Communication Association’.
URL: http://www.nikkostrom.com/publications/interspeech2015/strom_interspeech2015.pdf.
- Therneau, T. & Atkinson, B. (2018), *rpart: Recursive Partitioning and Regression Trees*. R package version 4.1-13.
- Thomas, J. (2017), *autoxgboost: Automatic tuning and fitting of xgboost*. R package version 0.0.0.9000,
URL: <https://github.com/ja-thomas/autoxgboost>.
- Vanschoren, J., van Rijn, J. N., Bischl, B. & Torgo, L. (2013), ‘Openml: Networked science in machine learning’, *SIGKDD Explorations* **15**(2), 49–60.
URL: <http://doi.acm.org/10.1145/2641190.2641198>.
- Wang, J., Xu, J. & Wang, X. (2018), ‘Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning’, *arXiv preprint arXiv:1801.01596*.
- Wright, M. N. & Ziegler, A. (2017), ‘ranger: A fast implementation of random forests for high dimensional data in C++ and R’, *Journal of Statistical Software* **77**(1), 1–17.
URL: <https://cran.r-project.org/web/packages/ranger/index.html>.
- Xiao, H., Rasul, K. & Vollgraf, R. (2017), ‘Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms’.
- Zhong, Z., Zheng, L., Kang, G., Li, S. & Yang, Y. (2017), ‘Random erasing data augmentation’, *CoRR* **abs/1708.04896**.
URL: <http://arxiv.org/abs/1708.04896>.